A DENOTATIONAL ENGINEERING OF PROGRAMMING LANGUAGES

To make software systems reliable
and user manuals clear, complete, and consistent

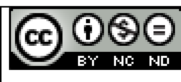
A book in statu nascendi
(a working version)

Andrzej Jacek Blikle
Piotr Chrząstowski-Wachtel
Janusz Jabłonowski
Andrzej Tarlecki

It always seems impossible
until it's done.

Nelson Mandela

Warsaw, August 19th, 2024



„A Denotational Engineering of Programming Languages" by A.J. Blikle, P. Chrząstowski-Wachtel, J. Jabłonowski, and A. Tarlecki has been licensed under a Creative Commons: Attribution-Noncommercial-NoDerivatives 4.0 International. For details see: https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode

Acknowledgments

The writing of this book started in 2013. Since then, the following of our colleagues have contributed to it with their remarks (ordered historically): Stanisław Budkowski, Antoni Mazurkiewicz, Marek Ryćko, Bogusław Jackowski, Ryszard Kubiak, Paweł Urzyczyn, Marek Bednarczyk, Wiesław Pawłowski, Krzysztof Apt, Jarosław Deminet, Katarzyna Wielgosz, Marcin Stańczyk, Albert Cenkier, Włodzimierz Drabent.

Our special thanks go additionally to:

- Stefan Sokołowski, who pointed out some inconsistencies of our earlier treatment of *assertions* and *invariants,*
- Jan Madey, who came with many bibliographical comments,
- Krzysztof Apt, who contributed with numerous substantial remarks and suggestions to the program-correctness model,
- Radosław Waśko, who pointed out some inconsistencies in the definitions of quantifiers for yokes.

Nelson Mandela’s quotation on the front page has been taken from
https://www.brainyquote.val/authors/nelson_mandela.

Contents

Acknowledgments 2
1 INTRODUCTION 8
1.1 What motivated us to write this book? 8
1.2 Building mathematically correct programs 8
1.3 Designing languages with mathematical semantics 9
1.4 What is in the book? 11
1.5 What differentiates our approach from the others? 12
1.6 Where are we on the way from user expectations to an executable code? 13
2 METASOFT AND ITS MATHEMATICS 14
2.1 Basic notational conventions of MetaSoft 14
2.1.1 General rules 14
2.1.2 Sets 15
2.1.3 Functions 16
2.2 Tuples 19
2.3 Partially ordered sets 20
2.4 Chain-complete partially-ordered sets 21
2.5 A CPO of formal languages 23
2.6 Equational grammars 24
2.7 A CPO of binary relations 25
2.8 A CPO of denotational domains 29
2.9 Abstract errors 31
2.10 Two three-valued propositional calculi 32
2.11 Data algebras 34
2.12 Many-sorted algebras 35
2.13 Abstract syntax and reachable algebras 38
2.14 Ambiguous and unambiguous algebras 42
2.15 Algebras and grammars 44
2.16 Abstract-syntax grammar is LL(k) 49
3 AN INTUITIVE INTRODUCTION TO DENOTATIONAL MODELS 51
3.1 How did it happen? 51
3.2 From denotations to syntax 53
3.3 Why we need denotational models of programming languages? 54
3.4 Five steps to a denotational model 55
3.5 Six steps to the algebra of denotations 57
3.6 Lingua as a strongly-typed language 58
4 DATA, TYPES, VALUES AND YOKES 59
4.1 Data 59
4.2 The types of data 62
4.3 Typed data 66
4.4 Yokes 70
4.5 Values, references, objects, deposits and types 74
5 CLASSES AND STATES 78
5.1 Classes intuitively 78
5.2 Classes formally 80
5.3 Stores and states 81
5.4 Two regimes of handling items 83

5.4.1 An overview ... 83
5.4.2 Usability regime ... 84
5.4.3 Visibility regimes ... 85

6 DENOTATIONS ... 90

6.1 The carriers of the algebra of denotations ... 90
6.2 Identifiers, class indicators and privacy statuses ... 91
6.3 Programs and their segments ... 92
6.4 Expressions ... 93

6.4.1 Value expressions ... 93
6.4.2 Yoke expressions ... 97
6.4.3 Type expressions ... 98
6.4.4 Reference expressions ... 100

6.5 Instructions ... 100

6.5.1 Signatures of constructors ... 100
6.5.2 Assignment instructions ... 101
6.5.3 Structural instructions ... 101

6.6 Methods ... 104

6.6.1 An overview of methods ... 104
6.6.2 Signatures and parameters ... 106
6.6.3 Imperative pre-procedures ... 106

6.6.3.1 An intuitive understanding ... 106
6.6.3.2 Creating imperative pre-procedures ... 107
6.6.3.3 A static compatibility of parameters ... 108
6.6.3.4 Passing actual parameters to a procedure ... 109
6.6.3.5 Returning the references of reference parameters ... 113
6.6.3.6 Calling an imperative procedure ... 116

6.6.4 Functional pre-procedures ... 116

6.6.4.1 Creating functional pre-procedures ... 116
6.6.4.2 Calling functional procedures ... 117

6.6.5 Object pre-constructors ... 117

6.6.5.1 Object constructors versus imperative procedures ... 117
6.6.5.2 Creating an object pre-constructor ... 118
6.6.5.3 Calling object constructors ... 119

6.7 Declarations ... 120

6.7.1 An overview of declarations ... 120
6.7.2 Declarations of variables ... 121
6.7.3 Declarations of classes — a basic constructor ... 122
6.7.4 Class transformers ... 123

6.7.4.1 The signatures of constructors ... 123
6.7.4.2 Adding an abstract attribute to the objecton of a class ... 124
6.7.4.3 Adding a concrete attribute to the objecton of a class ... 125
6.7.4.4 Concretizing abstract attributes and adding concrete attributes ... 126
6.7.4.5 Adding a type constant to a class ... 126
6.7.4.6 Adding a method constant to a class ... 127
6.7.4.7 Composing transformers sequentially ... 128

6.7.5 Enrichments of covering relations ... 128
6.7.6 The openings of procedures ... 129

7 SYNTAX AND SEMANTICS ... 131

7.1 An overview of syntax derivation ... 131
7.2 Abstract syntax ... 132

7.2.1 General remarks 132
7.2.2 Identifiers, class indicators and privacy statuses 132
7.2.3 Type expressions 132
7.2.4 Value expressions 132
7.2.5 Reference expressions 133
7.2.6 Yoke expressions 133
7.2.7 Instructions 134
7.2.8 Declarations 134
7.2.9 Openings of procedures 134
7.2.10 Class transformers 134
7.2.11 Preambles of programs 134
7.2.12 Programs 134
7.2.13 Declaration-oriented carriers 134
7.2.14 Signatures 135

7.3 Concrete syntax 135

7.3.1 General remarks 135
7.3.2 Identifiers, class indicators and privacy statuses 135
7.3.3 Type expressions 136
7.3.4 Value expressions 136
7.3.5 Reference expressions 136
7.3.6 Yoke expressions 136
7.3.7 Instructions 137
7.3.8 Declarations 137
7.3.9 Openings of procedures 137
7.3.10 Class transformers 138
7.3.11 Preambles of programs 138
7.3.12 Programs 138
7.3.13 Declaration-oriented carriers 138
7.3.14 Signatures 138

7.4 Colloquial syntax 139

7.4.1 New constructor of attribute declarations 139
7.4.2 The list of colloquial domains 140

7.5 Semantics 140

7.5.1 The ultimate semantics of Lingua 140
7.5.2 Why do we need a denotational semantics? 143

8 SEMANTIC CORRECTNESS OF PROGRAMS 145

8.1 Historical remarks 145
8.2 A relational model of nondeterministic programs 146
8.3 Iterative programs 147
8.4 Procedures and recursion 150
8.5 Three concepts of program correctness 150
8.6 Partial correctness 154

8.6.1 Sequential composition and branching 154
8.6.2 Recursion and iteration 156

8.7 Weak total correctness 159

8.7.1 Sequential composition and branching 159
8.7.2 Recursion and iteration 161

9 VALIDATING PROGRAMMING 165

9.1 Languages of validating programming 165

9.2 Conditions 167
9.2.1 General assumptions about conditions 167
9.2.2 Value-oriented conditions 168
9.2.3 Cov-oriented conditions 168
9.2.4 Value-, type- and reference-oriented conditions 171
9.2.5 Procedure-oriented conditions 172
9.2.6 Assertions and specified programs 173
9.2.7 Algorithmic conditions 175
9.3 Metaconditions 176
9.3.1 Basic categories of metaconditions 176
9.3.2 Properties of metapredicates 178
9.3.3 Metaconditions associated with programs 179
9.4 Metaprogram constructions rules 182
9.4.1 A birds-eye view on a metaprogram development 182
9.4.2 Correctness-preserving modifications of metaprograms 183
9.4.3 Universal rules 184
9.4.4 Rules for metadeclarations 185
9.4.4.1 Variable declarations 186
9.4.4.2 Enrichment of a covering relation 186
9.4.4.3 Class declarations 186
9.4.5 The opening of procedures 187
9.4.6 Rules for metainstructions 188
9.4.6.1 Rules for composed instructions 188
9.4.6.2 Rules for assignment instructions 188
9.4.6.3 Rules for imperative procedure calls 190
9.4.6.4 The case of recursive imperative procedures 192
9.4.6.5 The case of functional procedures 193
9.4.6.6 Jaco de Bakker paradox in Hoare's logic 194
9.5 Transformational programming 195
9.5.1 First example 195
9.5.2 Changing the types of data 201
9.5.3 Adding a register identifier 202
10 RELATIONAL DATABASES INTUITIVELY 205
10.1 Preliminary remarks 205
10.2 Basic values and their types 205
10.3 Creating tables 207
10.4 Databases and subordination relation between tables 209
10.5 Instructions of table modification 210
10.6 Transactions 211
10.7 Queries 213
10.8 Views 215
10.9 Cursors 216
10.10 The client-server environment 217
11 A DENOTATIONAL MODEL FOR DATABESES: Lingua-SQL 218
11.1 Lingua-SQL as an enrichment of Lingua 218
11.2 Data, types, values and states 219
11.2.1 Basic data, types and values 219
11.2.2 Columns, their yokes and types 220
11.2.3 Labeled rows and row yokes 222
11.2.4 Tables and their types 223

11.2.5 Databases and their subordination relations 225
11.2.6 States 229
11.3 The algebra of denotations 230
11.3.1 Replicated denotations and their constructors 230
11.3.2 The carriers of the algebra of denotations 231
11.3.1 Constructors of primitive denotations 233
11.3.2 Expressions 233
11.3.2.1 Categories of SQL expressions 233
11.3.2.2 Basic-type expressions 233
11.3.2.3 Row expressions 234
11.3.2.4 Column-yoke expressions 234
11.3.2.5 Row-yoke expressions 235
11.3.2.6 Column-marking expressions 235
11.3.2.7 Column-type expressions 235
11.3.2.8 Table-header expressions 236
11.3.2.9 Table-type expressions 236
11.3.3 Declarations of table variables 236
11.3.4 Instructions 237
11.3.4.1 Row-oriented table instructions 237
11.3.4.2 Column-oriented table instructions 239
11.3.4.3 Transactions 243
11.3.4.4 Queries 243
11.3.4.5 Instructions modifying integrity constraints 243
11.3.4.6 Cursors 244
11.3.4.7 Views 244
12 AN EXERCISE WITH A DENOTATIONAL CONCURRENCY 245
12.1 An overview of our model of concurrency 245
12.2 Bundles of computations 247
12.2.1 Abstract nets and quasinets 247
12.2.2 Nets of the bundles of computations 248
12.2.3 Strong total correctness of bundles 251
12.2.4 Temporal quantifiers 253
12.3 Petri nets and trace languages 253
12.3.1 Trace languages of Antoni Mazurkiewicz 253
12.3.2 Trace languages and Petri Nets 258
12.3.3 Petri nets redefined 261
12.3.4 Petri nets with data flow 264
12.4 Building a language of concurrent programs 267
12.4.1 General assumptions about the language 267
12.4.2 A case study of a structured constructor 268
13 INDICES AND GLOSSARIES 272
13.1 References 275
13.2 Index of terms and authors 280
13.3 Index of notations 283
14 WHAT REMAINS TO BE DONE 272
14.1 Computer-aided program development 272
14.2 Computer-aided language design 273
14.3 Techniques of writing user manuals 274
14.4 Programming experiments 274
14.5 Building a community of Lingua supporters 274

1 INTRODUCTION

1.1 What motivated us to write this book?

It is a well-established engineering practice that designing a new product starts from a blueprint supported by mathematical calculations. Both provide a mathematical warranty that the future functionality of the product will satisfy the expectations of its designer and user.

In software engineering, the situation is different. In the place of blueprints and calculations, programmers develop their codes starting from a contract between a future user and an IT producer, usually written in a more or less technical language but without mathematical rigor. The future target code is developed from such a contract through a sequence of steps, where this contract is "translated" to increasingly more and more technical descriptions and ends up with a compilable code. Although all these descriptions have a professional character, they still do not offer a mathematical precision comparable to, e.g., differential equations of a bridge designer.

As a consequence of this situation, large parts of budgets for program developments are spent on testing, i.e., removing errors introduced at the coding stage. Since testing may only discover some faults but never guarantee their absence, the non-discovered bugs are passed on to users to be removed later under the name "maintenance". In several cases, this situation led to spectacular catastrophes practically always — to many nuisances for users. The latter are, therefore, forced to accept a disclaimer like the following one:

There is no warranty for the program to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair, or correction.

Is it possible that a producer of a car, a dishwasher, or a building could request such a disclaimer from their clients? Why, then, is the software industry an exception? In our opinion, one of the causes of this situation is a lack of adequate mathematical tools for software engineers to guarantee the functional reliability of their products.

Of course, we are aware that mathematical tools will not solve all problems of software engineering, as they are not solving all problems in other industries. At the same time, however, we are convinced that there is a need for "more mathematics" in software production. This book proposes two sets of mathematical tools for software engineers: one addressed to language designers and the other to programmers. We believe that taking responsibility by software engineers for their products should be possible to the same extent as in remaining industries.

1.2 Building mathematically correct programs

The issue of mathematically provable program correctness appeared for the first time in a work by Alan Turing [87] published in conference proceedings *On High-Speed Calculating Machines*, which took place at Cambridge University in 1949. Later, that subject was investigated for several decades under the name of "proving program correctness", but the developed methods never became a standard tool for the software industry. Consequently, many people concluded that research in this field is not worth the effort. A particularly explicit formulation of this opinion we found in a monograph, *Deductive Software Verification* [2], published in 2016:

For a long time, the term formal verification was almost synonymous with functional verification. In the last years, it became more and more clear that full functional verification is an elusive goal for almost all application scenarios. Ironically, this happened because of advances in verification technology: with the advent of verifiers, such as KeY, that mostly cover and precisely model industrial languages and that can handle realistic systems, it finally became obvious just how difficult and time-consuming the specification of the functionality of real systems is. Not verification, but specification is the real bottleneck in functional verification.

We strongly believe that the failure to prove program correctness in practical scenarios has two primary sources.

The first is an obvious observation that proofs of theorems are usually longer than the theorems themselves. Therefore, proofs of program correctness may contain thousands, if not millions of lines, which makes "hand-made proofs" unrealistic. Additionally, fully formalized proofs for "practical" programs are hardly possible due to the lack of formal semantics of the languages in which they have been written.

The second cause is even more critical — programs that are supposed to be proven correct are usually incorrect! Consequently, correctness proofs are regarded as methods of identifying errors in programs. Besides, the order, first a program and then the proof of its correctness, may seem natural for mathematicians — first a hypothesis and then its proof — but is somewhat awkward for engineers, who first prepare blueprints and calculations and only then build "their bridges".

To our knowledge, the inadequacy of the approach of first writing a program and only then trying to prove its correctness was pointed out for the first time by Edsger Dijkstra in 1976 in his book *A Discipline of Programming* [51] where he writes:

Between the lines (of his book) the reader may have caught a few more general messages. The first message is that it does not suffice to design a mechanism of which we hope that it will meet its requirements, but we must design it is such a form that we can convince ourselves — and anyone else for that matter — that it will, indeed, meet its requirements. And, therefore, instead of first designing a program and then trying to prove its correctness, we develop correctness proof and program hand in hand (our emphasis). (In actual fact, the correctness proof is developed slightly ahead of the program: after having chosen the form of the correctness proof, make the program so that it satisfies the proof's requirements.)

Dijkstra formalized this idea using his *weakest preconditions*. A little later (1997 – 1981), Andrzej Blikle published a few papers [25] - [28] technically different from Dijkstra's approach but in a similar spirit. In recent years, Dijkstra's ideas have been implemented by the authors of Dafny Environment [???] and their followers (cf. Tabea Bordis et al. [41]) under the name correct-by-construction.

In Dijkstra's approaches, the authors tacitly assume that their proposed program constructors are sound. They do not prove this soundness since their languages lack mathematical semantics. In our approach, we first show how to build programming languages with fully mathematical semantics and then how to develop sound program-construction rules for such languages.

1.3 Designing languages with mathematical semantics

By a mathematical semantics of a programming language, we shall mean a function that assigns meanings to programs. Since the 1970s, many researchers have started to believe that such semantics, to be "practical," must be compositional, i.e., that the meaning of a whole must be a composition of the meanings of its parts. Later, such semantics were called *denotational* — the meaning of a program is its *denotation* — and for about two decades, researchers investigated the possibilities of defining denotational semantics for existing programming languages. The two most complete (although not fully formalized) such semantics were written in 1980 for Ada [16] and CHILL [42] in a metalanguage VDM [15]. A little later, in 1987, Andrzej Blikle described a denotational semantics of a subset of Pascal [29]. In the latter case, the metalanguage was MetaSoft [29], primarily based on VDM.

Unfortunately, none of these attempts created a denotational semantics of a widely used programming language. In our opinion, this situation was caused by the fact that for most language designers, the *meaning*

of a program is a compiler's behavior generated by the execution of this program. Also, programmers, when they explain to each other what a given piece of syntax "means", describe what a compiler will "do" during the execution of this syntax. This machine-oriented understanding of program meanings has led to syntaxes that are not "suitable" for giving them denotational semantics. Technical arguments supporting this opinion will be offered later in Sec. Sec. 6 and 7.

Independent of these problems, many researchers have found denotational models technically too complicated. Let us quote just one such opinion expressed by Cliff Johns in his book *Understanding Programming Languages* [63]:

Denotational semantics is mathematically elegant but requires some fairly sophisticated mathematical concepts in order to describe programming languages of the sort that are used to build real applications.

Such opinions about denotational semantics were associated with their early-stage technicalities. One was a *jump instruction* **goto**, and the other — self-applicable procedures that can take themselves as parameters (Algol 60, see [7]). The former had led to *continuations* [84], the latter to *reflexive domains* [83]. Continuations were counterintuitive and reflexive domains — mathematically fairly complicated. Fortunately, although these mechanisms were considered necessary in the 1960s, they were abandoned ten years later.

In our approach, we use neither continuations nor reflexive domains. The idea of denotational semantics without these mechanisms was described by Andrzej Blikle and Andrzej Tarlecki in a joint paper [39] in 1983.

Besides the resignation of "historical" technicalities of denotational semantics, we use in our approach an idea proposed by Andrzej Blikle in [30] that in developing a programming language, we should start from its denotations DEN and derive from it a syntax SYN later. He proved that, in this case, a denotational semantics

DS : SYN → DEN

always exists and, additionally, is unique. Formally, SYN and DEN constitute many-sorted algebras (Sec. 2.12), and the associated semantics DS is a (unique) homomorphism between them. As it turns out, there is a simple method — to a large extent algorithmizable — of deriving syntax from (the description of) denotations and, later, the semantics for both of them.

To illustrate our method, we designed a virtual programming language, **Lingua.** At the level of data structures, it includes booleans, integers, reals, texts, records, arrays, and their arbitrary combinations plus objects. It is equipped with a relatively strong mechanism of user-definable data types on the one hand and object types, i.e., classes, on the other. Control structures available in **Lingua** include structural instructions and multi-recursion procedures. The language has a complete error-reporting mechanism, by which we mean that every run-time error (except infinite looping) is signalized by an error message. Of course, errors will not occur in correct programs.

At the end of the book, we show how to enrich **Lingua** by two following mechanisms:

1. an API for SQL,
2. concurrency at the level of simple Petri nets,

Of course, **Lingua** is not developed/described in all detail since, in such a case, the book would hardly be readable. Our exercise with **Lingua** only illustrates a language-designing method that (hopefully) may be used in some future to design and implement practical programming languages.

Nevertheless, an experimental interpreter of **Lingua** has been developed by a group of students attending courses given commonly by Andrzej Blikle and Alex Schubert at the Department of Mathematics, Informatics, and Mechanics of Warsaw University in the academic years 2019/20 and 2020/21.

Once we have a language with denotational semantics, we can define program-construction rules and prove their soundness. Our construction rules were sketched for the first time in [25]. In the present book, they are developed for **Lingua**. Technically, the rules are used to build so-called *metaprograms* that syntactically include their specifications. Program construction rules guarantee that if we combine two or more correct programs into a new program or transform a correct program, we get a correct one again. Therefore, the

correctness proofs of programs are implicit in how they are developed and in the soundness proofs of construction rules.

1.4 What is in the book?

As mentioned in the preceding sections, the book contains many thoughts developed during 1960-1990 but later abandoned. One of the teams developing these ideas was working at the Institute of Computer Science of the Polish Academy of Sciences, and two of us — A.Blikle and A.Tarlecki — enjoyed working there. We then created a semi-formal metalanguage called **MetaSoft** [29] dedicated to formal definitions of programming languages.

Sec.2 introduces general mathematical tools used later in describing our basic model. In particular, they include:

1. a formal, but not formalized, definition of **MetaSoft**,
2. fixed-point theory in partially ordered sets,
3. the calculus of binary relations,
4. formal-language theory,
5. fixed-point domain equations based on so-called *naive denotational semantics* (rather than Scott and Strachey's reflexive domains),
6. many-sorted algebras,
7. abstract errors as tools for the description of error-handling mechanisms,
8. three-valued predicate calculi of McCarthy and Kleene,
9. equational grammars (equivalent to Chomsky's grammars and BNF's),
10. syntactical algebras based on equational grammars,
11. a short half-formal reminder of LL(k) grammars.

It should be emphasized in this place that Sec.2 may discourage less mathematically oriented readers. These readers may skip reading this section, maybe except Sec. 2.1 where notational conventions are explained. All mathematical tools and facts mentioned there are necessary to prove the mathematical soundness of our approach but are not prerequisites to understanding it.

Sec. 3 includes an intuitive description of our model and defines significant milestones to be passed through in its construction.

Sec. 4 is devoted to developing a general model of data structures and their types. Types, which are frequently regarded as sets of data, in our model are finitistic structures that indicate the "shapes" of the corresponding data. This approach allows for a definition of an algebra of types at a data level and an algebra of the denotations of type expressions at the level of denotations.

In Sec. 5 we introduce three fundamental concepts: a class, an object, and a state. We also discuss the visibility (privacy) issues of objects' attributes, typical of object-oriented mechanisms.

Sec. 6 describes the core of our model and is devoted to denotations. Here, we define the carriers and the constructors of an algebra of denotations. From a practical viewpoint, when we design an algebra of denotations, we make significant decisions on the tools offered by a language to programmers.

Once denotations are defined, we proceed in Sec. 7 to the derivation of syntax. Here, given a description of the algebra of denotations, we derive step-by-step three syntaxes: an abstract syntax, a concrete syntax, and a colloquial syntax. Formally, these syntaxes constitute algebras and are described by equational grammars. We also show how to derive a formal definition of a function of semantics once we are given an algebra of denotations and "its" algebra of syntax.

Sec.8 is devoted to an abstract theory of partial and total correctness of programs. This theory bases on an algebra of binary relations, making it universal for many programming languages.

In Sec. 9, starting from **Lingua**, we develop a corresponding language for validated programming, **Lingua-V**. In this case, however, we do not build a logic of programs — as C.A.R. Hoare [61], [5] or E. Dijkstra [50], [51] did — but we define a list of (proved) rules for the construction of correct programs.

Sec. Sec. 10 and 11 are devoted to enriching Lingua with SQL mechanisms. Since we do not expect our reader to be familiar with SQL details, we provide in Sec. 10 an intuitive introduction to its primary tools. In Sec 11, we give these tools a denotational model. As it turns out, extending **Lingua** to **Lingua-SQL** is more than just a simple enrichment of a source algebra of denotations by new carriers and constructors. It requires profound modifications and, therefore, offers a non-trivial example of enhancing a denotational model by new mechanisms.

Even more substantial changes to our model are necessary when, in Sec. 12, we introduce some concurrency mechanisms into **Lingua**. In this case, the flowchart-like structures of our programs are replaced by simple Petri nets enriched by the trace languages of Mazurkiewicz [72].

In Sec. 13, we included short remarks about what remains to be done in our project. In particular, we are talking about two computer-assisted work environments: one for the designers of languages using our method and another for programmers in such languages.

1.5 What differentiates our approach from the others?

Historically, the ideas of denotational engineering emerged from the early works of A. Blikle ([18] to [33]), A.Blikle with A. Mazurkiewicz [37], and A.Blikle with A. Tarlecki [39]. In turn, these works followed various approaches in this or another way. Below, we give references to the earliest papers on these approaches and to the significant contributions that followed:

- generative grammars of N. Chomsky ([44] in 1956, [45] in 1957, [46] in 1959, [47] in 1962, [55] in 1966, and [17] in 1971),
- denotational semantics of D. Scott's and Ch. Strachey's ([83] in 1971 and [84] in 1977),
- C.A.R Hoare's logic of programs (the founding paper [61] in 1969, and surveys [4] in 1981, [5] in 2020 and [6] in 2020),
- E. Dijkstra's total correctness of programs and the derivation of correct programs ([50] in 1968 and [51] in 1976),
- many-sorted algebras in computer science by J. A Goguen, J.W, Thatcher, E. G. Wagner and J. B. Wright ([58] in 1977),
- three-valued propositional calculus of J. McCarthy (cf. [74] in 1967),
- abstract errors in program's semantics originally introduced by Joe Goguen ([57] in 1978, [28] in 1981, [11] in 1984, [29] in 1987, [86] in 1988, and [31] in 1988), and later also investigated from a perspective of a Hoare's logic by J. V. Tucker and J. I. Zucker ([86] in 1988).

The main differences between our approach and other approaches to denotational semantics and program correctness are the following:

1. In the field of programming language design:
   - 1.1. our denotational models are based on set theory rather than D. Scott's and Ch. Strachey reflexive domains,
   - 1.2. the denotations of programs are state-to-state functions rather than continuation-to-continuation,
   - 1.3. denotations are developed in the first place, and syntax is derived from them later; the process of the derivation of syntax is highly algorithmizable,
   - 1.4. the idea of a colloquial syntax allows making syntax user-friendly without damaging mathematical rigor,
   - 1.5. our denotational models include:
     - 1.5.1. error-detection mechanisms supported by three-valued boolean expressions and predicates,
     - 1.5.2. objects and classes,
     - 1.5.3. SQL databases,
     - 1.5.4. simple Petri nets concurrency.
2. In the field of correct program development:

2.1. the soundness of program construction rules is proved on the grounds of a denotation semantics of the involved language,
2.2. the use of three-valued predicates enriches Dijkstra's total correctness approach by a clean-termination property.

1.6 Where are we on the way from user expectations to an executable code?

A virtual production line in a software factory may be seen — from a simplified perspective — as a sequence of the following actions:

1. the identification of user's needs (either in a dialog with the user or by market research),
2. the creation of a technical vision of future software architecture; this step usually includes many sub-steps where our intuitive image of the future software is gradually concretized,
3. the creation of a high-level program (coding) expected to be an adequate algorithmization of the output of 2.,
4. a compilation process,
5. running the compiled code by hardware.

Of course, each of these stages offers many error opportunities. Why, then, have we restricted our attention to step 3.?

The first answer is that our research experience predisposes us to tackle high-level program development more than the other steps. Besides, there is quite a lot of mathematical research available in this field (cf. Sec. 1.5).

Our second answer is based on the fact that compilers and hardware are much better tested today — due to their extensive use — than applications that are just being created, and therefore our choice of 3. before 4. and 5. seems partly justified.

Why then 3., rather than 1. or 2.? In this case, in addition to our former arguments, we can say that we do not know of languages used at early stages of software development that could be given mathematical semantics. At the same time, however, *domain-specific languages* may offer a promising perspective. Therefore, it may be vital to see to what extent our language development technique may open research opportunities in this area.

2 METASOFT AND ITS MATHEMATICS

From 1970 to 1990, Andrzej Blikle had been lecturing mathematical foundations of computer science to IT practitioners. In these cases, he frequently heard an objection that there is too much mathematics that software engineers have to swallow. Bosses of IT departments expected that their teams could be "trained" in that new mathematics within one weekend, well maximally two. In such cases, he tried to bring to their attention that future mechanical or electrical engineers attend two to five semesters of mathematics during their university studies. However, most of this mathematics was created at the borderline between the XIX and XX century and is oriented towards physics, astronomy, and classical engineering rather than informatics.

At the beginning of the second half of the XX century, mathematicians started to think about mathematical theories for computer science; some of the branches of mathematics earlier considered "unpractical" — such as set theory, mathematical logic, or abstract algebras — became their standard tools. A little later, new branches emerged: theory of abstract automata and formal languages, logic of programs, models of concurrent systems, and many others. Today, mathematical foundations of computer science embrace large and still fast-growing new branches of applied mathematics.

In the present section, we describe selected mathematical tools we shall use in the book. At the same time, we are conscious that going through this section may be pretty challenging for some readers. We may advise them only to slip over this math and possibly return to it when some technique used in subsequent sections will require a deeper justification.

2.1 Basic notational conventions of MetaSoft

2.1.1 General rules

MetaSoft is a semi-formal (i.e., not fully formalized) mathematical notation used in describing denotational models of programming languages. Each such model consist of three mathematical entities:

1. *Denotations* — the meanings of programs and their components such as expressions, instructions, declarations etc.
2. *Syntaxes* — programs and their components.
3. *Semantics* — a function that assigns denotations to syntaxes.

In the colloquial English of computer scientists denotations are most frequently confused with semantic. We can hear, e.g., that "the semantics of instructions are functions that modify memory states". In our approach we shall strictly distinguish between *denotations* that are the meanings of programs, syntaxes that are strings of characters used by programmers, and semantics that are function mapping syntaxes into denotation. We shall describe this fact by the following formula:

Sem : Syntaxes ⟼ Denotations

The notation used in this formula will be explained in Sec. 2.1.3. So far we may only notice that our mathematical formulas will be typeset in Arial, rather than in *Times New Roman Italic* as usual in mathematical texts. The reason of this decision is twofold:

1. in our texts we want to distinguish between an informal layer typeset in Times New Roman including its italic versions, and the layer of formulas,
2. as we are going to see, large and complex formulas that we shall use are better readable in Arial that in *Times Italic*.

In turn, to carefully distinguish between syntax and denotations, programs end their components will be typeset in Arial Narrow, e.g.,

while x > 100 **do** x := x-1 **od**

Additionally, since **while**, **do** and **od** are keywords, they are typeset in bold.

Another special property of **MetaSoft** is that (meta)mathematical variables that denote elements, sets and functions are frequently many-character symbols rather than single letters. This choice has been forced by the fact that in denotational models we use many more symbols than in "usual" mathematics, and therefore we should by giving them mnemotechnical forms. A typical example of a MetaSoft formula such as

ind : InsDen = WfState → WfState

is read as follows: the domain of instruction denotations InsDen is the set of partial functions that transform well-formed states into well-formed states. Elements of this domain will be denoted by ind possibly with prefixes or postfixes.

Another special notation concerns indexed variables. In traditional mathematics indices are written as subscripts, e.g., as a_i. Since this complicates typing and is not compatible with the syntaxes used in programs, we shall frequently write indices at the same level as an indexed symbol, e.g., as a-i. Of course, indices may be many-character symbols as well.

Our special notational conventions have one more justification. As we are going to see, the descriptions of denotational models in **MetaSoft** resemble programs, in particular codes of interpreters. In the future the writing of such descriptions should be assisted by dedicated editors. The outputs of these editors will then become inputs for generators of interpreters or compilers of corresponding programming languages.

Logical operators are given mnemotechnical names: **and**, **or**, **not**, tt, ff. The two last are logical constants "true" and "false". For quantifiers we shall use:

∀ — *general quantifier* (for all)
∃ — *existential quantifier* (there exists)

Instead of i = 1,…,n we shall write i = 1;n. By "iff" we shall mean "if and only if", and by "wrt" — "with respect to".

2.1.2 Sets

Symbol {} denotes an empty set and

{ele-1,…,ele-n} or {ele-i | i = 1;n}

denote finite sets of n elements. The fact that ele is, or is not, an element of a set of elements Element we shall write as

ele : Element   or respectively as   ele /: Element

For any sets A and B their inclusion will be written as

A ⊆ B

By

A | B and A∩B

we denote the union and the intersection of these sets . If FamSet is a family of sets then

U.FamSet

denotes the union of all the sets of this family. By

A x B

w denote the Cartesian product of sets. The expression:

A x B x C x D

denotes the set of tuples of the form (a, b, c, d), whereas the expression:

A x (B x C) x D

denotes the set of tuples of the form (a, (b, c), d), and analogously for other combinations of parentheses. For every $n \geq 0$ *the n-th Cartesian power* A^{cn} *of a set* A is the set of all n-tuples of the elements of A, i.e.:

$A^{c0} = \{()\}$ — the only element of that set is an empty tuple

$A^{cn} = \{(a\text{-}1,\ldots,a\text{-}n) \mid a\text{-}i : A\}$ — for $n > 0$

Given Cartesian powers, we can define two other operations:

$A^{c+} = U.\{A^{cn} \mid n > 0\}$ — Cartesian plus operation,

$A^{c*} = A^{c0} \mid A^{c+}$ — Cartesian star operation.

The set of all subsets of A and respectively of all finite subsets of A is denoted by

Sub.A

FinSub.A

The following notations shall be used for sets of relations and functions:

Rel.(A,B) — the set of all binary relations between A and B; i.e., the set of all subsets of A x B; more about binary relations in Sec.2.7,

$A \rightarrow B$ — the set of all *partial functions* from A to B, i.e., functions that do not need to be defined for all elements of A,

$A \longmapsto B$ — the set of all *total functions* from A to B, i.e., functions that are defined for all elements of A; notice that each total function is a partial function but not vice-versa,

$A \Longrightarrow B$ — the set of all *mappings* from A to B, i.e., functions defined for only a finite subset of A.

Following this notation by

$f : A \rightarrow B$

we mean that f is an element of the set $A \rightarrow B$, i.e. is a partial function from A to B, and analogously for other operators creating sets of functions. A is called the *domain* of f, and B is called its *range*. The use of colon ":" also explains why the traditional $a \in A$ we write as a : A.

2.1.3 Functions

For practical reasons, the value of a function fun for argument a shall be written as fun.a rather than fun(a). Why this is practical will be seen a little later. The expression

fun.a = ? (2.1-1)

means that fun is not defined for a. It does not mean that "?" is anything like an "undefined element". The expression fun.a = ? stands for

not (∃b)(fun.a=b)

Analogously

fun.a = !

stands for (∃b)(fun.a=b). For an arbitrary function

fun: A → B

and an arbitrary set C by the *truncation of* function f *to* C we shall mean:

fun truncate-to C = {(a, fun.a) | a : A ∩ C}.

The *domain of definedness* of function f is the set where f is defined, i.e.

dom.fun = {a | a : A **and** fun.a = !}

In the sequel we shall also use the notation

fun[a/?] = fun **truncate-to** (dom.fun – {a})

Another notation that will be used frequently comes from Haskell Curry and concerns many-argument function whose arguments are taken successively one after another. For instance, if

fun : A → (B → (C → (D → E))) (2.1-2)

then a value of such a function would be traditionally written as

((((fun.a).b).c).d)

but Curry writes it as

fun.a.b.c.d

which intuitively means that

- function f takes a as an argument and returns as a value a function fun.a that belongs to the set B → (C → (D → E)), and next
- function fun.a takes as an argument an element b and returns as a function fun.a.b that belongs to C → (D → E), etc.

This notation allows not only to avoid many parentheses but also to define function of "mixed" types like e.g.

fun : A → (B ↦ (C → (D ⇒ E))) or (2.1-3)
fun : (A → B) ↦ (C → (D ⇒ E))

Another simplifying convention allows to write

fun : A → B ↦ C → D ⇒ E (2.1-4)

instead of

fun : A → (B ↦ (C → (D ⇒ E))) (2.1-5)

The expression

fun : ↦ A (2.1-6)

means that fun is a zero-argument function with only one value that belongs to A. That value is denoted by

fun.()

About formulas in (2.1-2) to (2.1-6) we say that they describe *type* or *signatures* of corresponding functions. For instance we say that the function in (2.1-4) *is of the type*

A → B ↦ C → D ⇒ E

For every (possibly partial) function

fun : A → A,

by its *n-th iteration* where n = 0,1,2,…we shall mean the function

$fun^n : A \rightarrow A$

defined in the following way:

fun^0 is an *identity function* on A, i.e. $fun^0.a = a$ for every a : A,

$fun^n.a = fun.(fun^{n-1}.a)$ for n > 0.

In denotational descriptions of programming languages, we shall frequently use many-level *conditional definitions of functions* with the following scheme:

```
fun.x =
   pre-1.x ➔ val-1
   pre-2.x ➔ val-2
   …                                  (2.1-7)
   pre-n.x ➔ val-n
```

where each pre-i is a classical predicate, i.e., a total function with logical values tt or ff, and each val-i is some value. Formula (2.1-7) is read as follows:

if pre-1.x is true, then f.x = val-1 and otherwise,

if pre-2.x is true, then f.x = val-2 and otherwise,

…

Intuitively speaking, the evaluation of this function goes line by line and terminates at the first line where pre-i.x is satisfied. Of course, to make such a definition unambiguous, the disjunction of all predicates pre-i.x must evaluate to "true", which means that all these predicates must exhaust all cases. Our usual way to ensure this condition will be to write **true** for pre-n.x at the last line, which denotes a predicate, that is always true. It can also be read as "in all other cases".

In the scheme (2.1-7) we also allow the situation where, in the place of a val-i we have the undefinedness sign "?" which means that for x that satisfies pre-i.x, function f is undefined. This convention allows for conditional definitions of partial functions.

In conditional definitions we also use a technique similar to defining local constants in programs. For instance if fun : A x B ⟼ C we can write

```
fun.x =
   pre-1.x ➔ val-1
   let
      (a, b) = x
   pre-2.a ➔ val-2
   pre-3.b ➔ val-3
   …
```

which is read as: *let* x *be a pair of the form* (a, b). We can also use **let** in the following way:

```
fun.x =
   pre-1.x ➔ val-1
   let
      y = h.x
   pre-2.x ➔ fun-2.y
   pre-3.x ➔ fun-3.y,
   …
```

All these explanations are certainly not very formal, but the notation should be clear when it comes to its applications in concrete cases.

A finite total function fun : {a-1,…,a-n} ⟼ {b-1,…,b-n} defined by the formula:

fun.x =
    x=a-1 ➔ b-1
    x=a-2 ➔ b-2
    …
    x=a-n ➔ b-n
    true ➔ ?

shall be written as

[a-1/b-1,…,a-n/b-n] or alternatively as [a-i/b-i | i = 1;n].

The empty function will be denoted by [ ]. Let f : A → B and g : C → D. The *overwriting of* f *by* g is a function denoted by

f♦g : A|C → B|D

and defined in the following way:

(f♦g).x =
    g.x = ! ➔ g.x
    true ➔ f.x

In particular, if f.x=? and g.x=?, then f♦g.x=?. A special case of overwriting is an *update of a function* written as f[a-1/b-1,…, a-n/b-n] and defined by the formula

f[a-1/b-1,…, a-n/b-n].x =
    x = a-1 ➔ b-1
    …
    x = a-n ➔ b-n
    true ➔ f.x

We may also overwrite by an undefinedness:

(f[a-1/?,…, a-n/?]).x =
    x = a-1 ➔ ?
    …
    x = a-n ➔ ?
    true ➔ f.x

Given two sets of functions F and G, we may overwrite one set by the other:

F ♦ G = {f♦g | f : F, g : G}.

2.2 Tuples

An expression

(a-1,…,a-n) or alternatively (a-i | i=1;n)

denotes n-*tuple*. Consequently () denotes an empty tuple. The difference between tuples and finite sets is such that the order of elements in a tuple is relevant and repetitions are allowed, which is not the case for sets. E.g.

{a, b, c ,c} = {a, c, b } = {a, b, c} = … but
(a ,b, c, c) ≠ (a, c, c, b) ≠ (a, b, c)

where a, b and c are different with each other.

Tuples are used as mathematical models for several concepts and among others for pushdowns. In this case the following functions will be used later on in the book:

push.(b, (a-1,…,a-n)) = (b, a-1,…,a-n,) for n ≥ 0
pop.(a-1,…,a-n) = (a-2,…,a-n) for n ≥ 2

pop.(a)	= ()	
pop.()	= ()	
top.(a-1,…,a-n)	= a-1	for n ≥ 1
top.()	= ?	

An important operation on tuples is a *concatenation of tuples*:

(a-1,…,a-n) © (b-1,…,b-m) = (a-1,…,a-n, b-1,…,b-m).

We shall also use two predicates:

are-repetitions.(a-1,…,a-n)	= tt	iff there exist i ≠ j such that a-i = a-j
no-repetitions.(a-1,…,a-n)	= tt	iff there are no i ≠ j such that a-i = a-j

Tuples may also be regarded as functions from natural numbers into their elements i.e.

(a-1,…,a-n).i = a-i

In the sequel we shall also need a function that given a tuple, returns its length:

length.() = 0
length.(a-1,…,a-n) = n

and another function that given a tuple returns the set of its element:

elements.(a-1,…,a-n) = {a-1,…,a-n}

2.3 Partially ordered sets

Let A be an arbitrary set and let

⊑ : Rel(A,A)

be a binary relation in this set. Relation ⊑ is said to be a *partial order* in A, if for any a, b, c : A the following conditions are satisfied:

1. a ⊑ a — *reflexivity*
2. if a ⊑ b and b ⊑ c then a ⊑ c — *transitivity*
3. if a ⊑ b and b ⊑ a then a = b — *weak antisymmetricity*

If a ⊑ b, then we say that a is *smaller* than b or that b *is greater* than a. If additionally a ≠ b, then we say that a is *significantly smaller* than b or that b is *significantly greater* than a.

A pair (A, ⊑) is called a *partially ordered set* (abbr. POS), and the set A is called its *carrier*. The word "partial" indicates that not necessarily any two elements of A are comparable with each other. If

for any a and b either a ⊑ b or b ⊑ a,

then we say that ⊑ is a *total order*.

Of course, every total order is partial but not vice versa. An example of a partial order which is not total is the inclusion of sets. Such POS is called *set-theoretic POS*.

Let B be a subset of a partially ordered set A and let b : B. In this case

- b is called a *minimal element* in B, if there is no a : B such that a ⊑ b and a ≠ b
- b is called the *least element* in B, if for any a : B holds b ⊑ a,
- b is called a *maximal element* in B, if there is no a : B such that b ⊑ a and a ≠ b,
- b is called the *greatest element* in B, if for any a : B holds a ⊑ b.

There exist partially ordered sets without a minimal element and sets where there is more than one such element. However, if there is the least element in a set, then it is the unique minimal element and analogously for maximal and greatest elements.

An *upper bound* of B is such an element of A, which is greater than any element of B. Notice that an upper bound of a set does not need to belong to that set, but if it does, then it is the greatest element of the set.

If the set of all upper bounds of B has the least element, then this element is called the *least upper bound* of B[1]. If a two-element set {a, b} has the least upper bound, then we denote it by

$a \vee b$

In a set-theoretic POS, the least upper bound of a family of sets is the set-theoretic union of that family. This, of course, also concerns a family of two sets.

2.4 Chain-complete partially-ordered sets

Let (A, ⊑) be a partially ordered set. By a *chain* in that set we mean any sequence of elements of A:

a.1, a.2, a.3, …

such that a.i ⊑ a.(i+1). Here, for a change we write a.i instead of a-i. Note that in this case a may be regarded as a function with natural-number arguments. If the set of all elements of a chain has the least upper bound, then it is called the *limit* of that chain and is denoted by:

lim.(a.i | i = 1,2,…)

A POS is said to be *chain-complete partially ordered set* (abbr. CPO) if:

1. every chain in A has a limit,
2. there exists the least element in A.

This least element we shall denote by Φ.

A total function f : A ⟼ A is said to be *monotone* if a ⊑ b implies f.a ⊑ f.b and we say that it is *continuous* if the following two conditions are satisfied:

1. for any chain (a.i | i = 1,2,…) the sequence (f.(a.i) | i = 1,2,…) is also a chain,
2. if the former has a limit, then the latter has a limit as well and

   lim.(f.(a.i) | i = 1,2,…) = f.(lim.(a.i | i = 1,2,…)).

As is easy to see, every continuous function is monotone, which follows from the fact that

if a ⊑ b then lim(a, b, b, b, …) = b.

Continuous functions satisfy a so called *Kleene theorem* (see [64]) — which we shall frequently use in our applications.

Theorem 2.4-1 *If* f *is continuous in a chain-complete set, then the set of all solutions of the equation*

x = f.x (2.4-1)

is not empty and contains the least element defined by the equation

$Y.f = lim.(f^n.Φ \mid n = 0,1,2,…)$ ■

Proof of that theorem is very simple:

$f.(Y.f) = f.(lim.(f^n.Φ \mid n = 0,1,2,…)) = lim.(f^n.Φ \mid n = 1,2,…) = lim.(f^n.Φ \mid n = 0,1,2,…)$.

The last equality follows from the fact that $f^0.Φ = Φ$, hence adding $f^0.Φ$ to the chain, does not change its limit. The property that Y.f is the least fixed point follows from the fact that for any other fixpoint X.f , Φ ⊑ X.f and from the monotonicity of f we have $f^n.Φ ⊑ X.f$ hence $lim.(f^n.Φ \mid n = 0,1,2,…)$. ■

[1] The greatest lower bound is defined in an analogous way but we will not need this concept in the book.

The equation (2.4-1) is called a *fixed point equation* and its solution Y.t — the *least fixed point of function* f. It is the least solution of the equation (2.4-1), but in the sequel we will call it simply *the solution* since other solutions will not be concerned.

The concept of a one-argument continuous function may be simply generalised to functions of many arguments. We say that

$$f : A^{cn} \mapsto A \tag{2.4-2}$$

is *continuous wrt to its first element*, if for any tuple (a.1,…,a.(n-1)) the function

g.a = f.(a, a.1,…,a.(n-1))

is continuous. In an analogous way we define the continuity of f with respect to any other of its arguments.

A many-argument function (2.4-2) is called *continuous* if it is continuous in all of its arguments.

As we are going to see soon, continuous functions are fundamental for our applications since due to Kleene's theorem we can recursively define sets and functions. Such definitions will most frequently have the form

x.1 = f-1.(x.1,…,x.n)
…
x.n = f-n.(x.1,…, x.n)

Of course, every such set of equations may be regarded as one equation

X = f.X

in a POS over a Cartesian product A.1 x … x A.n where

f.(x.1,…,x.n) = (f-1.(x.1,…,x.n),…, f-n.(x.1,…,x.n))

and where the order is defined component-wise, i.e.

(a.1,…,a.n) ⊑(n) (b.1,…,b.n) iff a.i ⊑ b.i for i = 1;n.

As is easy to show, if all A.i's are chain-complete, then their Cartesian product is chain-complete wrt the above order. Besides, if all f-i are continuous, then f is continuous, as well.

As turns out, fixed-point sets of equations with continuous functions may be transformed (and reduced) in a way analogous to the case of algebraic equations. It is expressed by two theorems due to Hans Bekić [12] and Jacek Leszczyłowski [67].

Theorem 2.4-2 *If* f, g : A x A ⟼ A *are continuous, then the set of equations*

a = f.(a, b)
b = g.(a, b)

is equivalent to

a = f.(a ,b)
b = g.(f.(a, b), b) ■

Theorem 2.4-3 *If* f, g : A x A ⟼ A *are continuous, then the set of equations*

a = f.(a ,b)
b = g.(a, b)

is equivalent to

a = h.b
b = g.(a, b)

where h *is a function that to every* b *assigns the least fixed point of* f.(x, b) regarded as a one-argument function of x running over the set A. ■

As we are going to see, the theory of fixed-point equations in CPO is an important tool for writing recursive definitions of sets and of functions in denotational models.

2.5 A CPO of formal languages

Grammars of *natural languages* such as English, Polish or French may be regarded as algorithms allowing to check which sentences are grammatically correct and which are not. In this spirit Noam Chomsky[2] has developed in early 1960. his model of *generative context-free grammars* or simply *context-free grammars* (see [43] − [47]). Formal languages generable by such grammars have been called *context-free languages*.

Although his model turned out to be not wite adequate for natural languages, it was successfully applied to programming languages. In the early years for Algol 60 and Pascal, later for ADA and CHILL and many other languages. These applications contributed to a rapid development of their theory. The first internationally recognized monography on that subject was written in 1966 by Seymour Ginsburg [55], and the first Polish monography in 1971 by Andrzej Blikle [17]. A year later, Andrzej Blikle has published a paper on *equational grammars* [19], which are equivalent, in a sense, to context-free grammars.

This section contains a short introduction to context-free languages in the context of equational grammars which are discussed in Sec. 2.6.

Let A be an arbitrary finite set of symbols called an *alphabet*. By a *word* over A, we mean every finite tuple over A, including the empty tuple (). Traditionally words are written as sequences of characters, e.g., accbda.

Since words are tuples (of characters) we can apply to them the operation of concatenation defined in Sec. 2.2. E.g.

abdaa © eaag = abdaaeaag

Every set L of words over A is called a *formal language* (or simply a *language*) over A. By Lan.A we denote the family of all languages over A and by {} — an empty language (empty set). If P and Q are languages, then their *concatenation* is the language defined by the equation:

P © Q = {p © q | p:P **and** q:Q}.

As we see, by © we denote not only a function on words but also on languages. If it does not lead to ambiguities, P © Q is written as PQ. Since concatenation is an associative operation, we can write PQL instead of (PQ)L or P(QL). We shall also assume that concatenation binds stronger than set-theoretic union, hence instead of

(P © Q) | (R © S)

we shall write

PQ | RS.

It is also easy to see that concatenation is left- and right-distributive over the union, i.e.

(P | Q) R = PR | QR
R (P | Q) = RP | RQ

The n-th *power of a language* P is defined recursively:

P^0 = { () }

$P^n = P © P^{n-1}$ for n > 0

---

[2] Noam Chomsky — an American linguist, philosopher and political activist. Professor of linguistics at Massachusetts Institute of Technology, creator of the concept of generative grammars.

We shall also use two operators called respectively (language-theoretic) *plus* and *star*:

$P^{+} = U.\{P^{i} \mid i > 0\}$

$P^{*} = P^{+} \mid P^{0}$

Hence for an alphabet A, the set A^{+} is the set of all non-empty words over A, and A^{*} is the set of all words over A. Languages over A are subsets of A^{*}.

Note the difference between L^{*} and L^{c*}. Whereas the former is a set of words over the alphabet of L, the latter is a set of tuples of words of L.

Since the inclusion of sets is a partial order, (Lan.A, ⊆) is a CPO with empty language as the least element. As is easy to show, all operations on languages, which are defined above, plus the union of languages, are continuous. For any two languages, P and Q, their least upper bound is their union P | Q, and the limit of a chain of languages is the union of the elements of the chain.

2.6 Equational grammars

Since all the operations on languages defined in Sec. 2.5 are continuous, they can be used in fixed-point equations (Sec. 2.4) regarded as grammars. This idea is elaborated below.

Consider a simple example of a set of equations that defines the set of identifiers of a programming language. In our example we assume that identifiers always start from a letter:

Letter = {a, b, …, z}
Digit = {0, 1, …, 9}
Character = Letter | Digit
Suffix = {()} | Character © Suffix
Identifier = Letter © Suffix

Such sets of equations are called *equational grammars,* and their solutions (tuples of languages) are called *many-sorted languages*. In the above case, the defined many-sorted language is a tuple of five categories (sorts):

(Letter, Digit, Character, Suffix, Identifier).

The category Suffix has an auxiliary character since its only role is to express the fact that an identifier must start with a letter. Its equation can be eliminated in using the Theorem 2.3-2 and the Theorem 2.3-3. As is easy to prove

$\text{Suffix} = \text{Character}^{*}$

hence our grammar may be reduced to a more compact form

Letter = {a, b, …, z}
Digit = {0, 1, …, 9}
Identifier = Letter © (Letter | Digit)*

This grammar defines a many-sorted language, which consists of three categories — and therefore is different from the former — but defines the same set Identifier.

Let us now investigate equational grammars more formally (for details see [19]). Let A be an arbitrary non-empty finite alphabet and let

Fam ⊆ Lan.A

be an arbitrary family of languages over A. Let Pol.Fam denotes the least class of functions of the type:

$p : (\text{Lan.A})^{cn} \mapsto \text{Lan.A}$ where $n \geq 0$

which contains:

(1) all projections, i.e. functions of the form $f.(X.1,\ldots,X.n) = X.i$ for $i \leq n$,

(2) all functions with constant values in Fam,

(3) the union and concatenation of languages

and is closed over the composition (superposition) of functions.

Functions in Pol.Fam are called *polynomials over* Fam. Since all functions described in (1), (2) and (3) are continuous, and a composition of continuous functions is continuous, all polynomials are continuous.

By an *atomic language* over A we shall mean any one-element language {w}, where w : A*. Polynomials over an arbitrary set of atomic languages are called *Chomsky's polynomials*. Below a few examples of such polynomials:

p_1.(X,Y,Z) = {b}
p_2.(X,Y) = {b}
p_3.(X,Y,Z) = X
p_4.(X,Y,Z) = ({d}X{b}YY{c} | X) Z

Observe that for a complete identification of a polynomial we have to define its arity. This can be seen on the examples of p_1 and p_2 which are different although return the same language.

Polynomials which do not "contain" union — e.g., such as p_1, p_2, and p_3 — are called *monomials*. Since concatenation is distributive over union, every polynomial may be reduced to a union of monomials.

An *equational grammar* over an alphabet A is any fixed-point set of equations of the form:

X.1 = p-1.(X.1,…,X.n)
…
X.n = p-n.(X.1,…,X.n)

where all p-i's are Chomsky's polynomials over A. Since polynomials are continuous, this set of equations has a unique least solution (L.1,…,L.n). The languages L.1,…L.n are said to be *defined by our grammar*. We also say that they are *equationally definable*.

As has been proved in [19], the class of equationally-definable languages is identical with the class of *context-free languages* in the sense of Chomsky[3]. Such a class remains the same if we allow the operations "*" and "+" in polynomials and if polynomials are built over arbitrary equationally-definable languages. For proofs of all these facts, see [19].

Due to these facts in the sequel, equationally-definable languages will be called *context-free.*

2.7 A CPO of binary relations

Let A and B be arbitrary sets. Any subset of their Cartesian product A x B will be called a *binary relation* or just a *relation* between these sets. Hence

Rel(A,B) = {R | R ⊆ A x B}

is the set of all binary relations between A and B. Instead of writing (a,b) : R, we shall usually write a R b.

If A = B, then instead of Rel(A, A) we write Rel(A). For every A we define an *identity relation*:

[A] = {(a, a) | a:A}

By Ø, we shall denote the *empty relation*. Let now

Boolean = {tt, ff} — logical values
p : A → Boolean — a predicate

With every predicate, we assign an identity relation defined by

[3] Which means that for each equational grammar there exists an equivalent context-free grammar and vice versa.

$Id(p) = \{(a, a) \mid p.a = tt\}$

If $R : Rel(A,B)$, then

$dom.R = \{a \mid (\exists b : B)\ a\ R\ b\}$ — the *domain* of R
$cod.R = \{b \mid (\exists a : A)\ a\ R\ b\}$ — the *codomain* of R

Let $P : Rel(A,B)$ and $R : Rel(B,C)$. A *sequential composition* of P and R is a relation

$P \bullet R : Rel(A,C)$

defined as follows:

$P \bullet R = \{(a, c) \mid (\exists b : B)\ (a\ P\ b\ \&\ b\ R\ c)\}$

For every two relations, their composition always exists, although it may be an empty relation. As is easy to check $\bullet$ is associative i.e.

$(P \bullet R) \bullet Q = P \bullet (R \bullet Q)$

It is, therefore, legal to write $P \bullet R \bullet Q$. We shall also write PR instead of $P \bullet R$ whenever this does not lead to misunderstanding, and we shall assume that composition binds stronger than union, hence instead of

$(P \bullet R) \mid (Q \bullet S)$

we write

$PR \mid QS.$

In the sequel, the sequential composition of relations will be frequently applied in the case where the composed relations are function. In that case:

$(P \bullet R).a = R.(P.a)$

and therefore

$(P \bullet R \bullet Q).a = (P \bullet (R \bullet Q)).a = Q.(R.(P.a)))$

which means that in a sequential composition of functions, the composed functions are "executed" from left to right one after another.

Similarly as for languages, also for relations, we define the operations of power, plus and star:

$R^0 = [A]$ identity relation in over A
$R^n = RR^{n-1}$ for $n > 0$
$R^+ = U\ \{R^n \mid n > 0\}$
$R^* = R^+ \mid R^0$

The *converse relation* for R is defined as follows

$a\ R^{-1}\ b$ iff $b\ R\ a$

A relation R is called a *function*, if

for any a, b and c, if $a\ R\ b$ and $a\ R\ c$, ten $b = c$.

If R and R^{-1} are functions, then R is said to be a *convertible function* or a *one-one function*. If P and R are functions, then PR is also a function and

$(PR).a = P.(R.a)$

hence the composition of functions is their superposition.

The set of relations $Rel(A,B)$ constitutes a CPO with ordering by set-theoretic inclusion and the empty relation as the least element. All of the defined operations on relations are continuous. In the sequel we shall frequently refer to the following theorem:

Theorem 2.7-1 *For any* P, Q : Rel(A) *the least solutions of equations*

X = P | QX *and*
X = P | XQ

are respectively

X = Q*P *and*
X = P*Q

Moreover, if both P *and* R *are functions with disjoint domains, then both these solutions are also functions.*■

In this place, it is worth noticing that the set of partial functions

A → B

constitutes a chain-complete subset of (Rel(A,B), ⊆) that is closed under the composition of arbitrary functions and union of functions with disjoint domains. Of course, both these operations are continuous.

Due to these facts, functions can be defined by fixed-point (recursive) equations. Since A and B are arbitrary, this is also true for functions of type

$f : A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$

provided that appropriate constructors are defined. As a first example, consider a recursive definition of a function of an n-th power of number 2, i.e.[4].

power-of-two : Number → Number where Number = {0, 1, 2,…}
power-of-two.n = 2^n for an integer n ≥ 0

A recursive definition of that function is as follows:

power-of-two.n =
n = 0 ➔ 1
n > 0 ➔ power-of-two.(n-1) * 2

This definition written as a fixed-point equation in the set-theoretic CPO

(Number → Number, ⊆, [ ])

is as follows

power-of-two = zero ♦ (minus ● power-of-two) ● double

where

zero.n = [0/1]
minus.n = n-1 for n > 0
minus.0 = ?
double.n = 2 * n

Notice that all these functions are constants in our equation, hence the right-hand side of that equation represents a one-argument function in our CPO:

F.fun = zero ♦ (minus ● fun) ● double

Since, as is easy to prove, ♦ and ● are continuous on both arguments, our function F is continuous as well, and therefore — according to Kleene's theorem — the least solution of our equation is the limit (the union) of the following chain of functions:

F.{ } = zero = [0/1]

---

[4] Here we introduce a notational convention of VDM and MetaSoft where instead of using one-character symbols as in usual mathematics, we use many-character symbols for both sets and functions. As we are going to see later, this convention is practically a must in the case of denotational models where numbers of symbols goes into tens if not hundreds.

F.zero = zero ♦ (minus ● zero) ● duble = [0/1, 1/2]
F.(F.zero) = zero ♦ (minus ● F.zero) ● duble = [0/1, 1/2, 2/4]
…

Each element of that chain is a finite approximation of our function power-of-two.

Now let us consider a technically more complicated example of a two-argument function of power in the set of natural numbers:

power : Number x Number → Number

power.n.m =
m = 0 ➔ 1
m > 0 ➔ n * power.n.(m-1))

Also this definition can be expressed as a fixed-point equation in the CPO of binary relations:

Rel.(Number x Number, Number)

To see that, let us construct a fixed-point equation whose solution is the function:

power.(n, m) = n^m

regarded as a relation in our CPO. Let us start from the definitions of a certain operation of composition of functions

F, Q : Rel.(A x A, A). (2.7-1)

By the *composition of* F and Q *on the second argument,* we shall mean the relation

F ❷ Q = {((a, b), c) | (∃d) ((a, b), d) : F **and** ((a, d), c) : Q}

If F and Q are functions then

[F ❷ Q].(a, b) = Q.(a, F.(a, b))

The set of relations (2.7-1) is, of course, a CPO with set-theoretic inclusion. One can show that ❷ is continuous on both arguments. Since the limit of a chain is in our case the set-theoretic union, it is sufficient to show that ❷ is distributive over union on both arguments, which means that the following equalities hold (we assume that ❷ binds stronger than the union):

(F-1 | F-2) ❷ Q = F-1 ❷ Q | F-2 ❷ Q and

F ❷ (Q-1 | Q-2) = F ❷ Q-1 | F ❷ Q-2

Let then

((a, b), c) : (F-1 | F-2) ❷ Q

which means that there exists a d such that

((a, b), d) : (F-1 | F-2) **and** ((a, d), c) : Q

which means that there exist i and d such that

((a, b), d) : F-i **and** ((a, d), c) : Q

which means that there exists i such that

((a, b), c) : F-i ❷ Q

which means that

((a, b), c) : F-1 ❷ Q | F-2 ❷ Q

In this way, we have proved the inclusion

(F-1 | F-2) ❷ Q ⊆ F ❷ Q-1 | F ❷ Q-2

The proofs of the remaining three inclusions are analogous.

Since ❷ is continuous on both arguments the following fixed-point equation has the least solution:

power = zero | (minus ❷ power) ❷ times (2.7-2)

where:

zero(n, 0) = 1
minus.(n, m) = m-1 for m > 0, and for m = 0 this function is undefined
times.(n, m) = n*m

Since the set-theoretic union and our composition are both continuous in the CPO of relations (2.7-1), Kleene's theorem implies that the solution of (2.7-2) is the limit of the chain of relation

$P^0 \subseteq P^1 \subseteq P^2 \subseteq \dots$ (2.7-3)

which are functions defined in the following way:

P^0 = zero
P^{i+1} = (minus ❷ P^i) ❷ times for $i \geq 0$

This means that for every $i \geq 0$ function P^i is a partial function of power restricted to $m \leq i$:

P^i.(n, m) =
 m ≤ i ➔ m^i
 true ➔ ?

Since all these functions coincide on the common parts of their domains, the set-theoretic union of the chain (2.7-3) is a function, and it is the power function.

2.8 A CPO of denotational domains

One of the main tools of denotational models of software systems are sets traditionally called *domains*. These domains are most frequently defined using equations — sometimes fixed-point equations — based on functions that are listed below. Some of them have been already defined, but we recall their descriptions to have their full list in one place:

1. A | B — set-theoretic union
2. A ∩ B — set-theoretic intersection
3. A x B — Cartesian product
4. A^{cn} — Cartesian n-th power
5. A^{c+} — Cartesian +-iteration
6. A^{c*} — Cartesian *-iteration
7. FinSub.A — the set of all finite subsets
8. A ⟹ B — the set of all mappings including the empty mapping
9. A – B — set-theoretic difference
10. Sub.A — the set of all subsets
11. A → B — the set of all functions from A to B
12. A ⟼ B — the set of all total functions from A to B
13. Rel.(A,B) — the set of all relations between A and B

These operators may be used in not-recursive equations, e.g.:

State = Identifier ⟹ Data (2.8-1)
InsDen = State → State instruction denotations

where InsDen denotes a domain of the denotations of instructions, or in fixed point equations, e.g.:

Record = Identifier ⟹ Data (2.8-2)
Data = Number | Record.

Whereas definition (2.8-1) does not raises any doubts, in the case of (2.8-2) the situation is different. Since it is obviously a fixed-point equation we have to prove the continuity of ⟹ and |, but the continuity where? What is the CPO of domains? Set-theoretic inclusion is clearly its partial order, but what is the carrier?

Potentially that carrier should include all domains that we shall define in the future, hence something like the set of all sets. Unfortunately — as has been known since 1899[5] — such a set does not exist[6]. Despite this fact, our problem can be solved on the base of M.P. Cohn's [48] construction. As he has shown, for any set of sets B there exists a set of sets Set.B with the following properties:

1. all sets in B belong (as elements) to Set.B,
2. Set.B is closed under all our operations from 1) to 13),
3. Set.B is closed under unions of all denumerable families of its elements,
4. the empty set {} belongs to Set.B.

Following this construction, we choose as family B, the set of all "initial" domains that we shall use in our model, such as Boolean, Number, Identifier, Character, etc. Since (Set.B, ⊆) is a set-theoretic CPO, we can talk about the continuity of functions defined on sets in Set.B. As is easy to show operations from 1) to 8) are continuous, the difference of sets is continuous only on the left argument, and the remaining functions are not continuous, and therefore they cannot appear in fixed-point equations[7].

On this ground we can claim that the equation (2.8-2) a least solution)= defined by the theorem of Kleene (Sec.2.4). Records defined in that way may "carry" other records, but of a "lower-level" than themselves. At the end of that hierarchy, we have records carrying numbers. If however, we replace ⟹ by →, then we can't apply Kleene's theorem to it. More on that subject in Sec. 3.1.

The fact that non-continuous operators can't be used in a fixed-point context does not mean that they cannot be used in fixed-point equations "at all". For instance, our two sets of equations (2.8-1) and (2.8-2) can be legally combined into one:

```
dat : Data     = Number      |  Record                    (2.8-3)
rec : Record   = Identifier  ⟹ Data
sta : State    = Identifier  ⟹ Data
ind : InsDen   = State       → State
```

Although "as a whole" this is a fixed-point set of equations with one non-continuous operation, the recursion is present only in the second and the third equation where the operators are continuous. This set of equations is therefore "legal", and the existence of its least solution is guaranteed by Kleene's theorem.

---

[5] The concept that a set of all sets does not exist is tied to Russell's paradox, which was published by the British philosopher and mathematician Bertrand Russell in 19011. However, the paradox had already been discovered independently in 1899 by the German mathematician Ernst Zermelo. At the end of the 1890s, Georg Cantor, considered the founder of modern set theory, had already realized that his theory would lead to a contradiction. (credit to Bing)

[6] Formally speaking the attempt of constructing such a set leads to a contradiction. Indeed, let Z be the set of all sets. Let then Ze be the set of all sets that are their own elements and Zn — the set of all sets that are not their own elements. Since obviously Z = Ze | Zn, set Zn must belong to either Ze or Zn. The first case must be excluded since in that case Zn should belong to Zn. The second case is impossible either, since then Zn must not belong to itself. Intuitively speaking one can say that the collection of all sets is "to large to be a set".

[7] As an example, let us show that the operator → is not continuous. Let then $A_1 \subset A_2 \subset \ldots$be an arbitrary chain of mutually different sets, and let B be an arbitrary set. The sequence of domains $A_i \rightarrow B$ constitutes a chain but none of its elements contain a total function on the union UA_i, hence none of such functions belong to $U(A_i \rightarrow B)$, which means that $U(A_i \rightarrow B) \neq UA_i \rightarrow B$. In an analogous way we may show the non-continuity of the operators A ⟼ B and Rel.(A,B). Notice, however, that $U(A_i \Rightarrow B) = UA_i \Rightarrow B$, and similarly for the right-hand-side argument which means that ⟹ is continuous on both arguments.

The decision of not using non-continuous functions in fixed-point equations is due to the fact that Klenee's theorem is not satisfied in such cases, and, therefore, fixed-point equations do not correspond to recursion.

2.9 Abstract errors

For practically all expressions appearing in programs, their values in some circumstances can't be computed "successfully". Here are a few examples:

- expression x/y cannot be evaluated if the variables x or y have not been declared,
- expression x/y cannot be evaluated if the variables x or y have not been declared as numbers,
- expression x/y cannot be evaluated if the current value of y is zero,
- expression x+y cannot be evaluated if its value exceeds the maximal number allowed in current implementation; alternatively, if additions is a modulo operation, it will return an incorrect result,
- the value of the array expression a[k] cannot be computed if the variable a has not been declared as an array or if k is out of the domain of a,
- the query "Has John Smith retired?" cannot be answered if John Smith is not listed in the database.

In all these cases, a well-designed implementation should stop the execution of a program and generate an error message.

To describe such a mechanism formally, we introduce the concept of an *abstract error*. In a general case, abstract errors may be anything, but in our models, they will be words, such as, e.g.

'division by zero not allowed'.

They are closed in apostrophes to distinguish them from metavariables at the level of **MetaSoft**.

The fact that an attempt to evaluate the expression x/0 raises an error message can be now expressed by the equation:

x/0 = 'division by zero not allowed'

In the general case with every domain Data, we shall associate a corresponding domain with abstract errors

DataE = Data | Error

where Error is a universal set of all abstract errors that may be generated in course of the executions of our programs. This set will be regarded as a parameter of our denotational model. Now, every partial operation

op : Data.1 x … Data.n → Data,

whose partiality is *computable,*[8] may be extended to a total operation

ope : DataE.1 x … DataE.n ⟼ DataE

Of course ope should coincide with op wherever op is defined, i.e. if d.1,…,d.n are not errors and op.(d.1,…,d.n) is defined, then ope.(d.1,…,d.n) = op.(d.1,…,d.n).

An operation ope will be said to be *transparent for errors* or simply *transparent* if the following condition is satisfied:

if d.k is the first error in the sequence d.1,…,d.n, then ope.(d.1,…,d.n) = d.k

This rule indicates that the arguments of ope are evaluated one-by-one from left to right, and the first error (if it appears) becomes the final value of the computation.

The majority of operations on data that will appear in our models will be transparent. An exception are boolean operations discussed in Sec. 2.10.

---

[8] Partiality of a function f is computable, if there is an algorithm which for every element x can detect if f.x is defined or not. In the examples of this section all functions have computable partialities. However, it is a well-known fact, that in the general case the definedness of recursive functions is not computable. E.g. there is no algorithm which given a program, and a memory state, will check whether the execution of this program starting from this state will terminate. Consequently, we can't assume that any undefinedness will be signalized by an error message.

Error-handling mechanisms are frequently implemented in such a way that errors serve only to inform the user that (and why) program evaluation has been aborted. Such a mechanism will be called *reactive*. However, in some applications the generation of an error may initiate a recovery action. Such mechanisms will be called *proactive*.

As we shall see in the sequel, a reactive mechanism may be quite simply enriched to a proactive one. However, since the latter is technically more complicated, in the development of our example-language **Lingua**, except **Lingua-SQL**, we shall most frequently apply a reactive model. Proactive constructions are discussed in Sec. 6.5.3 and Sec. 11.3.4.3.

A well-defined error-handling mechanism allows avoiding situations where programs hang up without any explanation, or even worse — when they generate an incorrect result without warning the user (see Sec. 10.2).

2.10 Two three-valued propositional calculi

Tertium non datur — used to say ancients masters. Computers denied this principle.

In the Aristotelean logic, every sentence is either true or false. The third possibility does not exist. However, in the world of computers the third possibility is not only possible but inevitable. For instance the boolean expression x/y > 2 may evaluate to true, false or error if y = 0. Error is, therefore, the third logical value.

To describe the error-handling mechanism in boolean expressions the basic domain of two boolean values "true" and "false":

Boolean = {tt, ff}

must be enriched by a third element

BooleanE = {tt, ff, ee}

where ee stands for "error" or an undefinedness caused by an infinite execution. Infinite executions in boolean expressions may happen these expressions may include calls of functional procedures, which may loop.

We assume for simplicity that there is only one error element, since at the level of boolean expressions, all errors will be treated in the same way[9]. How are we handling non-computable undefinednesses will be seen a little later.

Let's observe now that the transparency of boolean operators would not be an adequate choice. To see that consider a conditional instruction:

if x ≠ 0 **and** 1/x < 10 **then** x := x+1 **else** x := x–1 **fi**

We would probably expect that for x=0, one should execute the assignment x:=x-1. If however, our conjunction would be transparent, then the expression

x ≠ 0 **and** 1/x < 10

would evaluate to 'division by zero not allowed', which means that our program would abort. Notice also that the transparency of **and** would imply

ff **and** ee = ee

which would mean that when an interpreter evaluates p **and** q, then it first evaluates both p and q — as in "usual mathematics" — and only later applies **and** to them. Such a mode is called an *eager evaluation*.

An alternative to it is a *lazy evaluation* where, if p = ff, then the evaluation of q is abandoned, and the final value of the expression is ff. In such a case:

ff **and** ee = ff

---

[9] Precisely speaking that is the case of a *reactive error elaboration*, i.e. all errors, except undefinedness, are displayed and abort program execution.

tt **or** ee = tt

A three-valued propositional calculus with lazy evaluation was described in 1961 by John McCarthy [74], who defined boolean operators as in Tab. 2.10-1.

or-m	tt	ff	ee
tt	tt	tt	tt
ff	tt	ff	ee
ee	ee	ee	ee

and-m	tt	ff	ee
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ee	ee

not-m	
tt	ff
ff	tt
ee	ee

Tab. 2.10-1 Propositional operators of John McCarthy

To see the intuition behind McCarthy's operators consider the expression p **or-m** q assuming that its arguments are computed from left to right[10]:

- If p = tt, then we give up the evaluation of q (lazy evaluation) and assume that the value of the expression is tt. Notice that in this case we possibly avoid entering an infinite computation.
- If p = ff, then we evaluate q, and its value becomes the value of the expression; it also means that if we enter an infinite execution, we remain in it. Here we also possibly avoid entering an infinite computation.
- If p = ee, then this means that the evaluation aborts or loops at the evaluation of p, hence q will never be evaluated. As a consequence, the final value of our expression must be the value of p. Note that in this place the infiniteness is "signalized" by itself, and, therefore, its non-decidability doesn't cause a problem.

The rule for **and** is analogous. It is to be emphasised that McCarthy's operators coincide with classical operators on classical values (grey fields in the table).

McCarthy's implication is defined in a classical way, i.e. by a combination of alternative and negation operators:

p implies-m q = (not-m p) or-m q

It is to be noted that not all classical tautologies remain satisfied in McCarthy's calculus. Among those that are satisfied we have[11]:

- associativity of disjunction and conjunction,
- De Morgan's laws

and among the non-satisfied are:

- **or-m** and **and-m** are not commutative, e.g., ff **and-m** ee = ff but ee **and-m** ff = ee,
- **and-m** is distributive over **or-m** only on the right-hand side, i.e.

  p **and-m** (q **or-m** s) = (p **and-m** q) **or-m** (p **and-m** s) however
  (q **or-m** s) **and-m** p ≠ (q **and-m** p) **or-m** (s **and-m** p) since
  (tt **or-m** ee) **and-m** ff = ff and (tt **and-m** ff) **or-m** (ee **and-m** ff) = ee

- analogously **or-m** is distributive over **and-m** only on the right-hand side,
- p **or-m** (**not-m** p) does not need to be true but is never false,
- p **and-m** (**not-m** p) does not need to be false but is never true.

---

[10] The suffix "-m" stands for "McCarthy" and is used to distinguish McCarthy's operators not only from classical ones but also from the operators of Kleene, which are discussed later.

[11] This claim is true only in the case of a single error element.

On the ground of McCarthy's calculus, we define in Sec. 6.4.1 the denotations of three-valued partial boolean expressions.

An alternative to McCarthy's propositional calculus is that of Kleene with operators defined in the following way:

or-k	tt	ff	ee
tt	tt	tt	tt
ff	tt	ff	ee
ee	tt	ee	ee

and-k	tt	ff	ee
tt	tt	ff	ee
ff	ff	ff	ff
ee	ee	ff	ee

not-k	
tt	ff
ff	tt
ee	ee

Tab. 2.10-2 Propositional operators of Steven Kleene

In this case

tt **or-k** ee = ee **or-k** tt = tt
ff **and-k** ee = ee **and-k** ff = ff

In Kleene's calculus whenever any argument of **or-k** is tt, then the result is tt, and analogously for **and-k**. Due to this assumption, we gain commutativity of both operators, but if we want to evaluate boolean expressions in this way, we had to compute both arguments of our operators "in parallel". This that is hardly acceptable, we use McCarthy's calculus in boolean expressions. If, however, we use a propositional calculus in proofs rather than in computations, then Kleene's calculus is more convenient. This is why we shall use it in conditions that describe properties of programs (see Sec. 9.2).

Another case where we shall use Kleene's calculus are special predicates called *yokes* (Sec. Sec. 4.4, 11.3.2.4 and 11.3.2.5), which are evaluated, but where procedures calls are not allowed.

2.11 Data algebras

Data types that are used in programs — such as integers, booleans, strings, arrays, lists, etc. — are usually associated with some operations on them. For instance, a data type of integers may be associated with the following arithmetical operations, and comparison predicates:

plus : IntegerE x IntegerE ⟼ IntegerE (2.11-1)
minus : IntegerE x IntegerE ⟼ IntegerE
times : IntegerE x IntegerE ⟼ IntegerE
divide : IntegerE x IntegerE ⟼ IntegerE
less : IntegerE x IntegerE ⟼ BooleanE
equal : IntegerE x IntegerE ⟼ BooleanE

where

int : IntegerE = Integer | Error
boo : BooleanE = {tt, ff} | Error

and where Integer is a set of integers representable in a current implementation.

A mathematical being that includes some sets and operations on them is called a *many-sorted algebra*. The sets in the algebra are called its *sorts*, and functions — its *constructors*.

As we are going to see later, an algebra of data usually includes more than one sort, and with each sort it includes some constructors. In our case we may add the following constructors of booleans:

or : BooleanE x BooleanE ⟼ BooleanE
and : BooleanE x BooleanE ⟼ BooleanE
not : BooleanE ⟼ BooleanE

Additionally, we may wish to add to our algebra zero-argument constructors that build some data "out of nothing":

create-zero : ⟼ IntegerE
create-one : ⟼ IntegerE
create-true : ⟼ BooleanE
create-false : ⟼ BooleanE

Such zero-argument operations are called *constants*. We may need constants in an algebra if we want our algebra ro have *reachable elements*, i.e. elements generable by constructors. Note that without integer constants we can't generate "the first integer". In turn, to do so we need only one constant create-one, since once we have 1 we can generate all other integers and booleans.

Sometimes, for technical reasons, we may wish to have "superfluous" constants, although usually not in the algebra of data. We will see such situations when building the algebras of denotations in Sec. 6.

Many-sorted algebras may be visualized graphically as in Fig. 2.11-1. For simplicity we included only some operation of the algebra and used shorter names of constructors

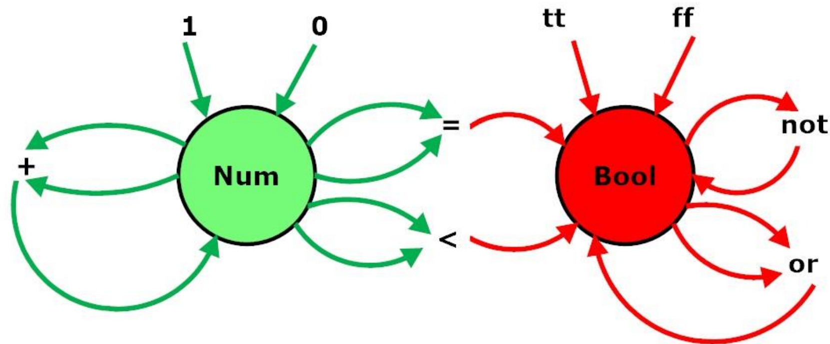


Fig. 2.11-1 Graphical representation of a two-sorted algebra

2.12 Many-sorted algebras

Our algebra discussed in Sec. 2.11, let's call it **AlgIntBoo**, is a two-sorted algebra and constitutes a particular case of many-sorted algebras. Such algebras will constitute one of our main tools in building denotational models, and therefore, we shall briefly introduce their theory in this section and sections 2.13, 2.14, and 2.15. Since this part of our book has an abstract mathematical character, we shall return for a while to traditional typesetting of indices as a_i rather than a.i.

By a many-sorted algebra we shall mean a tuple:

Alg = (Sig, Car, Fun, car, fun)

where

Sig = (Cn, Fn, ar, so) – is called the *signature of the algebra*,

Cn – is a finite set of words called the *names of carriers*; these names are usually called the *sorts of the algebra,*

Fn – is a finite set of words that are the *names of functions*; the functions themselves are called *constructors*

ar : Fn ⟼ Cn^{c*} – with every name of a function fn there is associated a finite (possibly empty) sequence of sorts

ar.fn = $(cn_1,\dots,cn_k)$

called the *arity* of fn[12]

so : Fn ⟼ Cn – to every name of a function fn the function so assigns a carrier name so.fn which is called *the sort of* fn,

Car – a finite set of *carriers*,

Fun – a finite set of total functions with arguments and values in carriers; these functions are called *constructors,*

car : Cn ⟼ Car – to every name cn of a carrier function car assigns a corresponding carrier car.cn,

fun : Fn ⟼ Fun – to every function name fn such that

$ar.fn = (cn_1,\dots,cn_k)$

so.fn = cn

the function fun assigns a total function

$fun.fn : car.cn_1 \times \dots \times car.cn_k \mapsto car.cn$

The concepts of *arity* and *sort* are applied not only to function names but also to the corresponding functions. Functions in the set Fun are traditionally called *constructors*. The tuple $((cn_1,\dots,cn_k), cn)$ that describes the arity and the sort of a constructor will be called the *signature* of that constructor.

Zero-argument constructors, i.e., constructors whose arity is an empty sequence, are called *constants* of the algebra. If f is such a constant, then we write

f : ⟼ Carrier

and the unique value of f is written as

f.()

It should be emphasized that all constructors of an algebra are total functions. In our approach this is possible due to the use of abstract errors (Sec. 2.9).

As we will see in the sequel, the signatures of many-sorted algebras have been introduced to describe the derivation of syntax from denotations in constructing programming languages. For concrete algebras, e.g., such as discussed in Sec.2.11, the signature is implicit in a corresponding set of formulas (2.11-1).

Two many-sorted algebras are said to be *similar* if they have the same signature. In the future, we shall frequently define concrete algebras by defining their carriers and constructors but without showing their signatures explicitly. In that case, we shall say that two algebras are similar if it is possible to construct a common signature for them.

Consider two algebras

$\mathbf{Alg}_i = (Sig_i, Car_i, Fun_i, car_i, fun_i)$ for i = 1,2

with signatures

$Sig_i = (Cn_i, Fn_i, ar_i, so_i)$ for i = 1,2

We say that Sig_2 is an *extension* of Sig_1 or that Sig_1 is a *restriction* of Sig_2, if

1. $Cn_1 \subseteq Cn_2$ and $Fn_1 \subseteq Fn_2$,
2. functions ar_2, so_2 coincide with ar_1, so_1 on Fn_1.

We say that algebra $\mathbf{Alg}_2$ is an *extension of algebra* $\mathbf{Alg}_1$, if

[12] The word „arity" comes from unary, binary, ternary etc.

1. Sig_2 is an extension of Sig_1,
2. $car_1.cn \subseteq car_2.cn$ for every sort $cn : Cn_1$,
3. $fun_2.fn$ coincides with $fun_1.fn$ on the appropriate carriers for every $fn : Fn_1$.

In other words, each (nontrivial) extension of an algebra results from that algebra by adding new carriers and/or new constructors and/or new elements to the existing carriers.

If $\mathbf{Alg}_1$ and $\mathbf{Alg}_2$ are similar, then we say that $\mathbf{Alg}_1$ is a *subalgebra* of $\mathbf{Alg}_2$ if:

1. the carriers of $\mathbf{Alg}_1$ are subsets of the corresponding carriers of $\mathbf{Alg}_2$,
2. the constructors of $\mathbf{Alg}_1$ coincide with constructors of $\mathbf{Alg}_2$ on the carriers of $\mathbf{Alg}_1$.

For every algebra there exists its unique subalgebra (maybe empty), called the *reachable subalgebra*, that includes only these elements of the algebra that can be generated by its constructors. If an algebra is identical with its reachable subalgebra, then it is said to be *reachable*.

Another important concept associated with many-sorted algebra are *many-sorted* homomorphism between them. By a *many-sorted homomorphism* from algebra $\mathbf{Alg}_1$ into a similar algebra $\mathbf{Alg}_2$ where we call a family of functions

$$H = \{h.cn \mid cn : Cn\}$$

whose elements — called *the components of that homomorphism* — map the elements of $\mathbf{Alg}_1$ into the elements of $\mathbf{Alg}_2$, hence

$$h.cn : car_1.cn \mapsto car_2.cn \quad \text{for all } cn : Cn$$

and where for every constructor name $fn : Cn$ such that

$$ar.fn = (cn_1,\dots,cn_n) \quad \text{where} \quad n \geq 0$$

and every tuple of arguments

$$(a_1,\dots,a_n) : car_1.cn_1 \times \dots \times car_1.cn_n$$

the following equality holds

$$h.cn.(fun_1.fn.(a_1,\dots,a_n)) = fun_2.fn.(h.cn_1.a_1,\dots,h.cn_n.a_n) \qquad (2.12\text{-}1)$$

In other words a homomorphic image of the value of a function $fun_1.fn$ from the first algebra with arguments $(a_1,\dots,a_n)$ equals the value of the corresponding function $fun_2.fn$ from the second algebra applied to the tuple of homomorphic images of the first tuple i.e. applied to $(h.cn_1.a_1,\dots,h.cn_n.a_n)$. Notice that for $n = 0$ the equality (2.12-1) has the form

$$h.cn.(fun_1.fn.()) = fun_2.fn.()$$

The fact that H is a homomorphism from $\mathbf{Alg}_1$ into $\mathbf{Alg}_2$ shall be written as:

$$H : \mathbf{Alg}_1 \mapsto \mathbf{Alg}_2$$

In the general case, homomorphisms do not map algebras <u>onto</u> algebras but <u>into</u> algebras, which means that not every element in $\mathbf{Alg}_2$ must be an image of an element form $\mathbf{Alg}_1$. For instance an identity homomorphism from integers to numbers

$$I2N : (Integer, 1, plus, minus) \mapsto (Number, 1, plus, minus)$$

is not "onto", whereas a homomorphism from integers into even integers

$$I2E : (Integer, 1, plus, minus) \mapsto (Even, 1, plus, minus)$$

defined by the equality $I2E.int = 2*int$ is "onto". In the general case a homomorphism $H : \mathbf{Alg}_1 \mapsto \mathbf{Alg}_2$ is called:

- a *monomorphism* — if all its components are one-to-one functions; e.g., I2N and I2E,
- an *epimorphism* — if all its components are "onto"; e.g., I2E
- an *isomorphism* — if it is both a monomorphism and an epimorphism; e.g., I2E.

Theorem 2.12-1 *For every homomorphism* $H : \mathbf{Alg}_1 \mapsto \mathbf{Alg}_2$, *the image of* $\mathbf{Alg}_1$ *in* $\mathbf{Alg}_2$, i.e., the restriction of $\mathbf{Alg}_2$ *to the images through* H of $\mathbf{Alg}_1$ *with the appropriate truncation of constructors of* $\mathbf{Alg}_2$ *constitutes a subalgebra of* $\mathbf{Alg}_2$. ■

Proof To prove our theorem, we have to show that the images in $\mathbf{Alg}_2$ of the carriers of **Alg1** are closed under the operations of $\mathbf{Alg}_2$. Let then $(b_1,\ldots,b_n)$ from $\mathbf{Alg}_2$, be the image of $(a_1,\ldots,a_n)$ in $\mathbf{Alg}_1$, i.e. let:

$$(b_1,\ldots,b_n) = (h.cn_1.a_1,\ldots,h.cn_n.a_n)$$

Let furthermore for some function name fn

$$fun_2.fn.(b_1,\ldots,b_n) = b$$

We have to show that b has a coimage in Alg_1. It is indeed the case since on the ground of (2.12-1):

$$fun_2.fn.(b_1,\ldots,b_n) = fun_2.fn.(h.cn_1.a_1,\ldots,h.cn_n.a_n) = h.cn.(fun_1.fn.(a_1,\ldots,a_n))$$

hence $h.cn.(fun_1.fn.(a_1,\ldots,a_n))$ is the coimage of b in $\mathbf{Alg}_1$. ■

An algebra, which is the image of a homomorphism, $\mathbf{Alg}_1 \mapsto \mathbf{Alg}_2$ is called *the kernel of the homomorphism* H in $\mathbf{Alg}_2$.

All our investigations about homomorphisms can be generalized to the case where the signatures of two algebras

$$Sig_i = (Cn_i, Fn_i, ar_i, so_i) \qquad \text{for } i = 1,2$$

are not identical but are *similar* in the sense that there exist two reversible functions of similarity

$$Sn \quad : Cn_1 \mapsto Cn_2$$
$$Sf \quad : Fn_1 \mapsto Fn_2$$

such that if

$$Sf.fn_1 = fn_2$$
$$ar_1.fn_1 = cn_{11},\ldots,cn_{1p}$$
$$ar_2.fn_2 = cn_{21},\ldots,cn_{2m}$$

then

$$p = m$$
$$Sn.cn_{1i} = cn_{2i} \quad \text{for } i = 1;p$$

In other words, two signatures are similar if they have the same number of carrier names and function names, and the corresponding function names have identical arities and sorts up to the names of carriers.

Now we can generalize the notion of the similarity of algebras: two algebras shall be called *similar* if their signatures are similar. For any fixed functions, Sn and Sf the concept of homomorphism, and the corresponding theorems remain valid for the generalized similarity.

2.13 Abstract syntax

Every signature

$$Sig = (Cn, Fn, ar, so)$$

unambiguously determines a certain algebra with that signature and with formal languages as carriers. This algebra is called *abstract syntax over signature* Sig and will be denoted by **AbsSy**(Sig)[13]. The elements of its carriers are words of a many-sorted formal language

[13] The idea of an abstract syntax regarded as a mathematical idealization of a syntax of a programming language appeared for the first time in papers of J. McCarthy [74] and P. Landin [66]. Abstract syntax was associated with abstract algebras by J.A. Goguen, J.W. Thacher, E.G. Wagner and J.B. Wright [58]. A little later A.Blikle [29] used that

{Lan.cn | cn : Cn}

defined by an equational grammar (see Sec.2.6) in a way described below.

To every carrier name cn we associate a language denoted by Lan.cn. The family (tuple) of all these languages is defined by an equational grammar where for every cn : Cn we have the following equation[14]:

$$\text{Lan.cn} = \{\text{fn}_1\} © \{(\} © \text{Lan.cn}_{11} © \{,\} © \ldots © \{,\} © \text{Lan.cn}_{1n(1)} © \{)\} \mid \quad (2.13\text{-}1)$$
$$\ldots$$
$$\{\text{fn}_k\} © \{(\} © \text{Lan.cn}_1 © \{,\} © \ldots © \{,\} © \text{Lan.cn}_{n(k)} © \{)\}$$

Here fn_i for i = 1;k are function names with

$\text{so.fn}_i = \text{cn}$

and

$\text{ar.fn}_i = (\text{cn}_{i1},\ldots,\text{cn}_{in(i)})$ for i = 1;k

We assume that if for a carrier name cn there is no function name fn such that so.nf = cn, then the corresponding language is empty, i.e. its defining equation is:

Lan.cn = {}

For every non-empty Lan.cn, its elements are words of the form

$\text{fn}_i(\text{w}_{i1},\ldots,\text{w}_{in(i)})$

i.e. of the form $\text{fn}_i © (© \text{w}_{i1} © \ldots © \text{wi}_{n(i)} ©)$ where © is the concatenation of words and

$\text{w}_{ik} : \text{Lan.cn}_k$.

As is easy to see, for every algebra **Alg** its abstract syntax algebra is reachable, although it may be empty if there are no constants in **Alg**.

Since abstract syntaxes are generated from signatures, they may be associated with arbitrary algebras (through their signatures). If **Alg** is an algebra with signature Sig, then **AbsSy**(Sig) will be called *the abstract syntax of algebra* **Alg**. For instance, if **AlgIntBoo** is the two-sorted algebra described in Sec.2.11 then the carrier of its abstract syntax are defined by the following equational grammar, where IntExp and BooExp are languages of integer expressions and boolean expressions respectively:

```
IntExp =                                        (2.13-1)
   0                      |
   1                      |
   plus(IntExp, IntExp)   |
   minus(IntExp, IntExp)  |
   times(IntExp, IntExp)  |
   divide(IntExp, IntExp)

BooExp =
   tt                     |
   ff                     |
   less(IntExp, IntExp)   |
   equal(IntExp, IntExp)  |
   or(BooExp, BooExp)     |
   and(BooExp, BooExp)    |
   not(BooExp)
```

concept in an attempt to give a formal semantics to a subset of Pascal . In his paper abstract syntax was technically understood in a slightly different way than here, but the idea was roughly the same.

[14] We assume, of course, that the commas "," and the parentheses "(" and ")" do not appear in the signature as constructors' names.

In this grammar, we use four notational conventions that we shall assume as standards for future use (cf. Sec. 2.1.1):

1. as already announced in Sec. 2.1.1, characters and words such as 0, 1, plus, (,) etc. that appear at the level of syntax are typeset in Arial Narrow, whereas IntExp and BooExp are typeset in Arial, since they are metavariables from the level of **MetaSoft**,
2. one-element sets are identified with their elements, i.e. instead of {a} we write a,
3. the values of zero-argument constructors are written without the empty tuples of arguments, i.e. we write 1 instead of 1.().
4. the concatenation sign © is omitted, e.g., instead of a © b we write a b,

Examples of a numeric and a boolean abstract-syntax expressions written in this style are the following:

- plus(plus(minus(1,0),1),plus(1,1))
- or(less(plus(plus(minus(1,0),1),plus(1,1)),plus(1,1)),ff)

As we see, the expressions of our languages do not contain variables and are written in a *prefix notation* where function symbols always precede their arguments. E.g., we write plus(1,1) instead of (1 plus 1). The latter style is called *infix-notation*.

In the syntactic algebra defined by our grammar, the elements of carriers are numeric and boolean expressions, respectively (without variables), and constructors correspond to constructor names from our signature. For instance, with a constructor name plus, we associate a constructor [plus] of the algebra **AbsSy**(Sig) defined by the equation

[plus].[num-exp$_1$, num-exp$_2$] = plus(num-exp$_1$,num-exp$_2$)[15]

This constructor, given two expressions num-exp$_1$ and num-exp$_2$ returns the expression of the form plus(num-exp$_1$,num-exp$_2$). E.g. given times(x,y) and plus(z,y) returns

plus(times(x,y),plus(x,y))

Now we can formulate a theorem which is fundamental for denotational models of programming languages.

Theorem 2.13-1 For every many-sorted algebra **Alg** with a signature Sig there is exactly one homomorphism H : **AbsSy**(Sig) ⟼ **Alg**. ■

Proof Every homomorphism H : **AbsSy**(Sig) ⟼ **Alg** must (from the definition) satisfy the equation:

H.cn.[fn(w$_1$, … , w$_n$)] = fun.fn.[H.cn$_1$.w$_1$,…,H.cn$_n$.w$_n$]

where

ar.fn = (cn$_1$,…,cn$_n$)
so.fn = cn
w$_i$: Lan.cn$_i$ for i = 1;n

Since every word in abstract syntax is of a unique (for it) form fn(w$_1$, … , w$_n$), the above equations (for all fn) define the family {H.cn | cn : Cn} in an unambiguous way. In the case of empty carriers of **AbsSy**(Sig) the corresponding components of H are empty. ■

The unique homomorphism from **AbsSy**(Sig) to **Alg** will be called *the semantics of abstract syntax*. For instance, if by {In, Bo} we denote the semantics of abstract syntax of **AlgIntBoo**, then this homomorphism maps boolean expression less(plus(1,1), times(1,1)) into boolean value ff:

Bo.[less(plus(1,1), times(1,1))] =
fun.less.(No.[plus(1,1)], No.[times(1,1)]) =
fun.less.(fun.plus.(No.[1],No.[1]), fun.times.([No.[1], No.[1])) =

[15] The meta-parentheses "[" and "]" are introduced in order to distinguish them from parentheses that belong to the defined language.

fun.less (fun.plus(1,1), fun.times(1,1)) =
ff

On the ground of theorems 2.12-1 and 2.13-1, in every algebra **Alg**, there is a unique subalgebra which is the kernel of the semantics of abstract syntax of **Alg**. That algebra is the (unique) reachable subalgebra of **Alg**. For instance, the reachable subalgebra of the algebra

(RealE, 1, plus, divide)

is the algebra of positive rational numbers

(PosRat, 1, plus, divide)

since only such numbers can be constructed from 1 in using plus and divide. Notice that if we remove 1 from this algebra, then its reachable subalgebra becomes empty and consequently its algebra of abstract syntax will be empty as well.

Theorem 2.13-2 For any two similar algebras $\mathbf{Alg}_1$ and $\mathbf{Alg}_2$, if Alg_1 is reachable, then there is at most one homomorphism

$H : \mathbf{Alg}_1 \mapsto \mathbf{Alg}_2,$

and if this is the case, then the image of Alg_1 in Alg_2 is reachable. ■

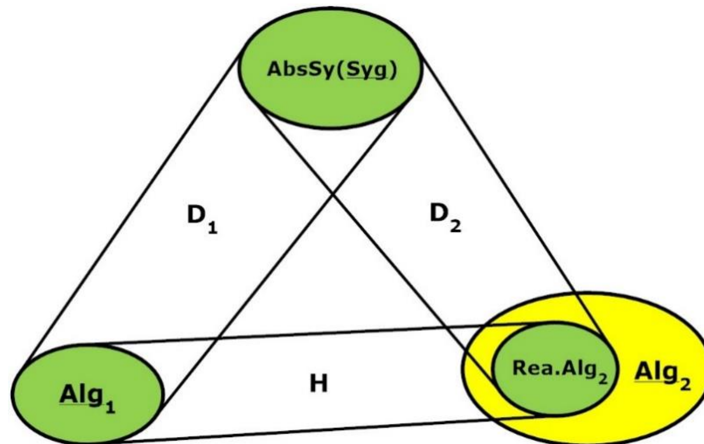


Fig. 2.13-1 Reachable algebras

Proof. The theorem and its proof are illustrated in Fig. 2.13-1. Since $\mathbf{Alg}_1$ and $\mathbf{Alg}_2$ are similar, they must have a common signature Sig and a common abstract syntax **AbsSy**(Sig). Therefore — on the ground of Theorem 2.13-1 — there exist two unambiguously defined semantics of abstract syntaxes

$D_1 : \mathbf{AbsSy}(Sig) \mapsto \mathbf{Alg}_1$ and
$D_2 : \mathbf{AbsSy}(Sig) \mapsto \mathbf{Alg}_2$

Now, if there exists a homomorphism $H : \mathbf{Alg}_1 \mapsto \mathbf{Alg}_2$, then the composition

$D_1 \bullet H : \mathbf{AbsSy}(Sig) \mapsto \mathbf{Alg}_2$

defined as the composition of their components is a homomorphism. Since D_2 is the unique homomorphism between these algebras, we have

$D_1 \bullet H = D_2,$

and since $\mathbf{Alg}_1$ is reachable, the above equation defines H unambiguously, because otherwise, we could define another homomorphism from **AbsSy**(Sig) into $\mathbf{Alg}_2$ which would contradict Theorem 2.13-1. This proves that the image of $\mathbf{Alg}_1$ in $\mathbf{Alg}_2$ is reachable. ■

As an immediate consequence of this theorem we have another theorem:

Theorem 2.13-3 For every nonempty algebra **Alg** over signature Sig the following claims are equivalent:

(1) **Alg** is reachable,
(2) every homomorphism of the type H : **Alg**$_1$ ⟼ **Alg** (for an arbitrary **Alg**$_1$) is onto,
(3) the semantics of abstract syntax D : **AbsSy**(Sig) ⟼ **Alg** is onto. ■

Proof Let **Alg** be reachable and let for some **Alg**$_1$ similar to **Alg** there exist a homomorphism

H : **Alg**$_1$ ⟼ **Alg**,

and let

D : **AbsSy**(Sig) ⟼ **Alg**$_1$

be the abstract-syntax semantics of **Alg**$_1$. In that case

D ● H : **AbsSy**(Sig) ⟼ **Alg**

is the abstract-syntax semantics for **Alg**, hence, since **Alg** is reachable, then D ● H must be *onto*, and therefore also H must be *onto*. Hence (1) implies (2). Now (3) follows from (2) as its particular case, and (2) implies (1) by the definition of reachability. ■

At the end of this section, one more useful theorem:

Theorem 2.13-4 An algebra has a nonempty reachable subalgebra if and only if it contains at least one zero-argument constructor. ■

Proof If there is a constant in the algebra, then it belongs to its reachable part, and hence, this part is not empty. If, however, such o constant does not exist, then in the grammar corresponding to that algebra, there are no constant monomials, and therefore all the carriers of abstract syntax are empty. Therefore the reachable part of **Alg** is an empty algebra. ■

Abstract syntaxes are, in general, not very convenient in practical programming, and therefore they are usually replaced by more user-friendly syntaxes historically called *concrete syntaxes*. In such a case, elements of abstract syntax correspond to *parsing trees* of concrete scripts (see, e.g. [3]).

2.14 Ambiguous and unambiguous algebras

An algebra **Alg** with a signature Sig is said to be *unambiguous* if its abstract-syntax semantics

D : **AbsSy**(Syg) ⟼ **Alg**

is a monomorphism, i.e., if for every carrier Car.cn of **Alg** and every element e of that carrier there is at most one word w : Lan.cn in the abstract syntax **AbsSy**(Syg) such that

D.cn.w = e

Algebras which are not unambiguous will be called *ambiguous*.

Algebras of denotations of programming languages are practically always ambiguous. For instance, the algebra **AlgIntBoo** described in 2.11 is ambiguous since, e.g., two different words plus(plus(1,1),1) and plus(1,plus(1,1)) correspond to the same number 3.

Now consider two algebras **Alg**$_1$ and **Alg**$_2$ with a common signature Sig hence also with a common abstract syntax **SkAbs**(Sig). Let

D$_1$: **SkAbs**(Sig) ⟼ **Alg**$_1$

D$_2$: **SkAbs**(Sig) ⟼ **Alg**$_2$

be two corresponding abstract-syntax semantics. Algebra **Alg**$_1$ is said to be *less (or equally) ambiguous than* algebra **Alg**$_2$, that we shall writer as

Alg$_1$ ≼ Alg$_2$

if the homomorphism D$_2$ is gluing not more than D$_1$ (Fig. 2.14-1), i.e., if for any two words w$_1$ and w$_2$ in abstract syntax that belong to the same carrier Car.cn the following implication holds:

if D_1.cn.w_1 = D_1.cn.w_2 then D_2.cn.w_1 = D_2.nn.w_2

Intuitively speaking, whenever an element of **Alg**$_1$ may be constructed in two different ways, the two ways lead to the same element in **Alg**$_2$.

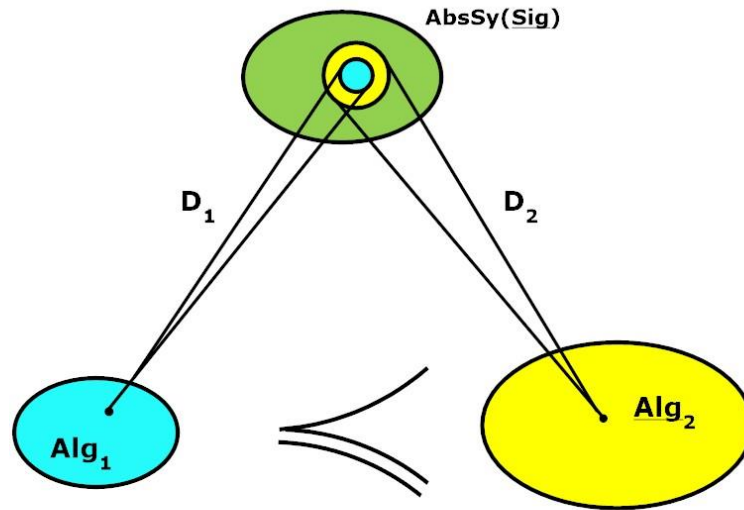


Fig. 2.14-1 Two ambiguous algebras

Ambiguous algebras play an important role in the theory of programming languages since, for the majority of existing languages, their algebras of concrete syntax — if formally described — would turn out to be ambiguous. To explain this fact assume that **AbsSy**(Sig) is defined by the grammar

IntExp = 0 | 1 | +(IntExp, IntExp),

Alg$_1$ is an algebra of infix expressions without parentheses defined by the grammar

IntExp = 0 | 1 | IntExp + IntExp

and **Alg**$_2$ is the algebra of integers. Let now D_1 replaces prefixes by infixes and removes parentheses.

Anticipating the considerations of Sec. 3, the algebra of numbers is the *algebra of denotations* (of meanings) for both our algebras of numeric expressions and the homomorphism D_2 is the *denotational homomorphism* (the *semantics*) of the algebra of abstract syntax. Now, we may ask, if there exists a denotational homomorphism

D_{12} : **Alg**$_1$ ⟼ **Alg**$_2$

from parentheses-free expressions into numbers.

To answer this question notice that for such algebras and their corresponding homomorphisms the following equalities hold:

D_1.[+(+(1,1),1)] = 1+1+1 D_2.[+(+(1,1),1)] = 3

D_1.[+(1,+(1,1)] = 1+1+1 D2.[+(1,+(1,1)] = 3

As we see D_1 is gluing not more than D_2. In "practical mathematics", hence also in programming languages, we frequently omit "unnecessary parentheses" when we deal with associative operations. The corresponding algebras are, in general, ambiguous, and therefore, the denotational homomorphism D_{12} need not exist. If however, they are not more ambiguous than the algebras of denotations, then such a homomorphism exist which follows from the following theorem:

Theorem 2.14-1 If **Alg**$_1$ and **Alg**$_2$ are similar and **Alg**$_1$ is reachable, then the (unique) homomorphism

D_{12} : **Alg**$_1$ ⟼ **Alg**$_2$ exists iff **Alg**$_1$ ≼ **Alg**$_2$. ■

This unique homomorphism may be constructed as (intuitively speaking) the composition of the inverse of D_1 with D_2, i.e.

$$D_{12} = D_1^{-1} \bullet D_2.$$

Although the inverse of D_1 maps the elements of $\mathbf{Alg}_1$ into sets of abstract expressions, yet all these expressions are mapped by D_2 into the same element of $\mathbf{Alg}_2$. For a formal proof of this theorem, see [32].

Of course, if D_1 is an isomorphism then $\mathbf{Alg}_1$ is "equally ambiguous" as $\mathbf{Alg}_2$, and therefore the homomorphism D_{12} exists.

2.15 Algebras and grammars

The first step in the process of programming-language construction consists in defining an algebra of denotations from which we derive a unique algebra of abstract syntax. Since the latter is usually not user-friendly, we transform it into a concrete syntax using a homomorphism that does not glue more than abstract-syntax semantics. Since in a user manual concrete syntax should be described by an equational grammar, we should raise a question, whether for any algebra of concrete syntax a corresponding grammar exists. To investigate this problem, we need the concepts of *a skeleton function*.

A function f on words over an alphabet A is said to be a *skeleton function* if there exists a tuple of words $(w_1,\ldots,w_k, w_{k+1})$ over A, called *the skeleton of this function* such that

$$f.(x_1,\ldots,x_k) = w_1x_1\ldots w_kx_nw_{k+1}$$

An example of a skeleton function may be

f.(exp-b,ins$_1$,ins$_2$) = **if** exp-b **then** ins$_1$ **else** ins$_2$ **fi**

The skeleton of this function is **(if, then, else, fi).** Notice that the function

f.(exp-b, ins$_1$, ins$_2$) = **if** exp-b **then** ins$_2$ **else** ins$_1$ **fi**

is not a skeleton function since the order of arguments on the left-hand side of our equation does not coincide with the order on its right-hand side.

In particular cases, a skeleton function may have more than one skeleton. E.g. the one-argument function

$$f : \{a\}^* \longmapsto \{a\}^*$$

defined by equation

$$f.(x) = x\ a$$

has two skeletons ((), a) and (a, ()), since it may be equivalently defined by the equation

$$f.(x) = a\ x$$

However, if we change the type of the function f to $f : \{a, b\}^* \longmapsto \{a, b\}^*$ without changing the defining equation, then this function has only one skeleton ((), a).

A many-sorted algebra will be called a *syntactic algebra* if it is a reachable algebra of words.

A syntactic algebra will be called a *context-free algebra* if all its constructors are skeleton functions. Of course, algebras of abstract syntax are context-free. As was shown in Sec. 2.13, for each such algebra, we can build an equational grammar that defines its carriers and constructors. Similarly, we may assign an equational grammar for any context-free algebra.

Theorem 2.15-1 For every context-free algebra, there is an equational grammar that generates is carriers. ■

The following theorem is also true:

Theorem 2.15-2 For every equational grammar there is a context-free algebra with carriers defined by that grammar. ■

Proof Let

$X_1 = pol_1.(X_1,\ldots,X_n)$
…
$X_n = pol_1.(X_1,\ldots,X_n)$

be an equational grammar with the (unique) solution $(L_1,\ldots,L_n)$. Assume that the polynomials of that grammar are expressed as unions of monomials. The corresponding algebra

Alg = (Sig, Car, Fun, car, fun),

is defined in the following way:

- Sig = (Nc, Nf, ar, so)
- Nc = $\{cn_1,\ldots,cn_n\}$ — carriers' names are arbitrary, but the number of these names must be equal to the number of equations in the grammar,
- Nf = $\{fn_1,\ldots,fn_m\}$ — function names are arbitrary, but the number of these names must be equal to the number of monomial occurrences in the grammar,
- ar and so are defined in that way, that they correspond to the arities and sorts of monomials in the grammar,
- Car = $\{L_1,\ldots,L_n\}$,
- Fun — the set of all monomials in our grammar,
- $car.cn_i = L_i$ for $i = 1,\ldots,n$

Notice now that every mononomial in our grammar is (from the definition) a Chomsky's mononomial (see Sec. 2.6), hence satisfies the equation:

$car.cn_i(x_1,\ldots,x_n) = \{s_1\}\ x_1 \ldots \{s_k\}\ x_k\ \{s_{k+1}\}$

This completes the definition of our algebra. Observe that the defined algebra is unique up to the names of carriers and constructors.

We can show that the carriers of **Alg** are closed wrt all its constructors and that the algebra is reachable. For proof see [32]. ■

Below is a simple example showing how to construct an algebra from a grammar. Consider the following grammar of a two-sorted language

Number = 1 | x | Number + Number

Boolean = Number < Number | Boolean & Boolean

For simplicity, curly brackets for function names have been dropped. The operations of our grammar are defined by the following equations (the symbols of concatenation © has been omitted as well) where n-exp and b-exp with indexes denote numerical and boolean expressions, respectively:

one.() = 1

variable.() = x

plus.($n\text{-}exp_1$, $n\text{-}exp_2$) = $n\text{-}exp_1$ + $n\text{-}exp_2$

less.($n\text{-}exp_1$, $n\text{-}exp_2$) = $n\text{-}exp_1$ < $n\text{-}exp_2$

and.($b\text{-}exp_1$, $b\text{-}exp_2$) = $b\text{-}exp_1$ & $b\text{-}exp_2$

An equational grammar is said to be *unambiguous* (resp. *ambiguous*) if the corresponding algebra is unambiguous (resp. ambiguous). Intuitively a grammar is ambiguous if there exists a word w that can be generated

by that grammars in two different ways[16]. These “different ways” are different elements of the abstract syntax that are coimages of w wrt the abstract-syntax semantics (see Sec. 2.13). For instance, the word 1+1+1 may be generated in two different ways:

plus(1,plus(1,1)

plus(plus(1,1),1)

As has been already mentioned, a concrete syntax of a programming language will be constructed as a homomorphic image of its abstract syntax. Since these syntaxes will be described by equational grammars, it is important to know which homomorphisms of syntactic algebras do not lead out of the class of context-free algebras.

Let us start with an example of a homomorphism that destroys the context-freeness of an algebra. Let **Alg** be a one-sorted algebra with the carrier $\{a\}^+$ and with two operations:

h.() = a

f.(x) = x a

This algebra is of course, context-free. Now consider a similar algebra with a carrier

$\{a^n b^n c^n \mid n = 1,2,\dots\}$

and constructors

h.() = abc

$f.(a^n b^n c^n) = a^{n+1} b^{n+1} c^{n+1}$

This algebra is not context-free since its carrier is a well-known example of a not context-free language (see [55]), but it is isomorphic with our former algebra where the corresponding isomorphism is:

$I.a^n = a^n b^n c^n$ for every $n \geq 1$

As is easy to see this isomorphism is not a skeleton function.

A homomorphism H between two syntactic algebras is called a *skeleton homomorphism* (we recall that since syntactic algebra are reachable, such a homomorphism, if exists, is unique (Theorem 2.13-3)) if for every constructor fun.fn of the source algebra, for which

so.fn = cn

$ar.fn = (cn_1,\dots,cn_n)$

there exists a skeleton $(s_1,\dots,s_{n+1})$, such that

$H.fn.(fun_1.fn.(x_1,\dots,x_n)) = s_1\ x_1 \dots s_n x_n s_{n+1}$

In other words, a homomorphic image of every constructor of the source algebra is a skeleton constructor in the target algebra.

Theorem 2.15-3 For every syntactic algebra **Alg** the following facts are equivalent:

(1) **Alg** is context-free,
(2) every homomorphism into **Alg** is a skeleton homomorphism,
(3) there exists a skeleton homomorphism into **Alg**.

For proof, see [19].

---

[16] The usability of ambiguous grammars also from the perspective of parsing was investigated in 1972 by A.V. Aho and J.D. Ullman in [3].

Let us consider now a simple example of a process of constructing a syntactic algebra for a given algebra[17]. Let the latter be a one-sorted algebra of numbers with three operations:

create-nu.1 : ⟼ Number

plus : Number x Number ⟼ Number

times : Number x Number ⟼ Number

The corresponding abstract syntax, denote it by Syn-0, is defined by the following grammar with only one equation, where Exp denotes a language of numerical expressions with constant values:

Exp = create-nu.1.() | plus(Exp, Exp) | times(Exp, Exp)

The first step on our way to final syntax consists in:

- replacing create-nu.1 by 1,
- replacing plus and times by + and *,
- replacing prefix notation by infix notation.

This step corresponds to the following homomorphism:

H.[create-nu.1.()] = 1

H.[plus(exp_1,exp_2)] = (H.[exp_1] + H.[exp_2])

H.[times(exp_1,exp_2)] = (H.[exp_1] * H.[exp_2])

This is of course a skeleton homomorphism and the corresponding context-free grammar is the following:

Exp = 1 | (Exp + Exp) | (Exp * Exp)

In the second and the last step of syntax construction we would like to allow dropping out "unnecessary parentheses", e.g. writing 1+1+1 instead of (1+(1+1)) and analogously for multiplication. Unfortunately this turns out to be impossible since each homomorphism which removes parentheses has to satisfy the equations:

H.[(exp_1 + exp_2)] = H.[exp_1] + H.[exp_2]
H.[(exp_1 * exp_2)] = H.[exp_1] * H.[exp_2]

but this would mean that it glues expressions with different denotations, e.g.

H.[(1+1)*(1+1)] = H.[((1+(1*1))+1)] = 1+1*1+1

Although H is a skeleton homomorphism, which implies that its target grammar

Exp = 1 | Exp + Exp | Exp * Exp

is context-free, the corresponding algebra is more ambiguous than the algebra of integers, hence a denotational semantics of this syntax into the algebra of numbers does not exist.

A known traditional way of solving this problem as e.g. in Algol ([7] and [80]) or in Pascal [62] consists in reconstructing the whole model of the language by introducing to the algebra of denotations and to the algebra of syntax three carriers Com (component), Fac (factor) and Exp (expression) and the following signature:

c-to-e	: Com	⟼ Exp	component to expression identically
+	: Exp + Com	⟼ Exp	addition
f-to-c	: Fac	⟼ Com	factor to component identically
*	: Fac * Com	⟼ Com	multiplication
1	: Fac	⟼ Fac	the generation of 1 as a factor
e-to-c	: Exp	⟼ Fac	expression to factor identically

[17] In more general terms such processes will be discussed in Sec. 3.4.

The corresponding grammar of abstract syntax is the following:

Exp = c-to-e(Com)| +(Exp, Com)
Com = f-to-c(Fac) | *(Fac, Com)
Fac = 1 | (Exp)

and for the first (isomorphic) transformed syntax:

Exp = (Com) | (Exp + Com)
Com = (Fac) | (Fac * Com)
Fac = 1 | (Exp)

In this grammar the names of identity functions have been omitted, which, however, does not destroy the unambiguity of the grammar, since these names appear in the elements of different carriers.

Now we can define a skeleton homomorphism that removes parentheses in each of three sorts of expressions:

E.[(val)] = val
E.[(val + exp)] = E.[exp] + S.[val]
C.[(fac)] = C.[fac]
C.[(fac * val)]= F.[fac] * C.[val]
F.[1] = 1
F.[(exp)] = (exp)

This leads to the following context-free grammar

Exp = Com | Exp + Com
Com = Fac | Fac * Com
Fac = 1 | (Exp)

This grammar may be also written in a direct way in using the constructor of iteration:

Exp = Com [+ Com]*	an expression is a sum of components
Com = Fac [* Fac]*	a component is a multiplication of factors[18]
Fac = 1 | (Exp)	a factor is a constant or an expression in parentheses

Observe that the parentheses-removal homomorphism is not an isomorphism, since it glues (1+(1+)) and ((1+1)+1) into 1+1+1 and similarly for multiplication. However it does not glue "to much" since addition and multiplication are associative. On the other hand from expression ((1+1)*(1+1)) it removes only external parentheses.

The denotational homomorphism for our grammar is now the following:

Se.[val] = Ss.[val]
Se.[exp + val] = Se.[exp] + Sc.[val]
Ss.[fac] = Sc.[fac]
Ss.[fac * val] = Sc.[fac] * Ss.[val]
Sc.[1] = 1
Sc.[(exp)) = Se.[exp]

Notice that the above equations express the school rules of priority of multiplication over addiction.

Commentary 2.15-1

The reader to whom we have promised that denotational models of programming languages will offer readable definitions may have some doubts in this moment. So far, the simple language of arithmetic expressions that is very well known to every ground-school student has been described in a rather complicated way and moreover using advanced mathematics. This, of course, requires a commentary.

First, what we can say to a student in a simple way, when "talking" to a computer, we have to express in a way

---

[18] Note the difference between the operation of multiplication *, e.g. as in 11* and the operation of the iteration of languages *, e.g. as in [+ Com]*.

appropriate for the interpreter. That "appropriate way" is a denotational homomorphism, which may be mapped one-to-one into a code of an interpreter.

Second, the discussed language serves only to illustrate the denotational method in an elementary example. The real advantage of the method will be appreciated (we hope) when we introduce more advanced programming mechanisms such as declarations, types, instructions, recursive procedures, objects, etc. whose definitions require more advanced mathematical tools.

Third, in writing a user's manual for our language, we may directly refer to our acquaintance with school mathematics by saying that numerical expressions can be written and are calculated in a "usual way", which frees us from the necessity of showing a grammar. However, as we shall see in Sec. 3.4 there are better solutions to that problem called *colloquial syntax*.

Two following lessons may be learned from our exercise:

First, the description of the simple operation of dropping out unnecessary parentheses requires rather complicated and not very intuitive grammar. Such a grammar is necessary for the implementor but not for the user, who can be simply informed that numerical expressions are written and understood in a "usual way".

Second, the idea of dropping parentheses came out only at the level of second syntactic algebra, when the two formers have already been defined. Therefore, to implement the parenthesis-free notation one has to restart the construction of the model from scratch. In our simple example, this does not lead to too much work, but in real situations, things may look different. To avoid such problems, one should think about syntax as early as on the level of the algebra of denotations. This, however, contradicts the philosophy "from denotations to syntax" and also ruins the principle that denotations should be constructed in a maximally simple way.

The above problems were investigated in [30], [32] and [40]. A solution suggested there consists in assuming that the programmer's syntax that will be called *colloquial syntax* does not need to be a homomorphic image of concrete syntax. In our example concrete syntax would be defined by the grammar:

Exp = 1 | (Exp + Exp) | (Exp * Exp)

and colloquial syntax — which allows for (although it does not force) the omission of parentheses — would be defined by the grammar:

Exp = 1 | (Exp + Exp) | (Exp * Exp) | Exp + Exp | Exp * Exp

Observe that the algebra of colloquial syntax is not only not-homomorphic to the former but is even not similar since it has a different signature (has more constructors).

Note, however, that it is easy to define a transformation that would map our colloquial syntax "back" into concrete syntax by adding the "missing" parentheses. Such a transformation will be called a *restoring transformation*. In practice, this approach leads to a user manual that contains a formal definition of concrete syntax (a grammar) plus an informal rule which says, e.g., that parentheses may be omitted in the "usual way"[19].

In the general case, a restoring transformation may be described formally or informally according to the complexity of colloquialization. Its formal definition is, however, always necessary for implementors who have to write a procedure that converts each colloquial program into its concrete version.

More on colloquial syntax in **Lingua** in Sec. 7.4.

In the end, one methodological remark seems necessary. Languages discussed in this section covered only expressions without variables. Such a case has, of course, no practical value, and it was chosen only to make examples of algebras and corresponding grammars possibly simple. Starting from Sec. 3.5 we shall discuss methods of constructing denotational models for more realistic languages.

2.16 Abstract-syntax grammar is LL(k)

Our equational grammars are equivalent to (well known in the literature) context-free grammars and the latter play an important role in the theory of the syntax of programming languages. Especially wanted context-free

[19] As we are going to see in Sec. Sec. 7.2 and 7.3 the situation may be a little more complicated.

grammars are LL(k) *grammars*, since their corresponding parsers are efficient and simple to build. To show that our abstract syntax grammars are LL(k), let's redefine this concept for equational grammars.

Consider an arbitrary equational grammar EG that generates a tuple of languages (Lan-1,…,Lan-n) over an alphabet Ter of characters called *terminals*. The elements of Lan-i's will be called *words*. Every equation of EG is the following formula

Syn-i = w-i1 | … | w-ip(i) for 1≤ i ≤ n (7.2-1)

where:

- Syn-i are metavariables corresponding to syntactic domains; we shall call them *nonterminals,*
- w-ij are metawords written over an alphabet Alp = Ter | {Syn-1,…,Syn-n},

Our grammar will be said to be *strongly prefixed,* if every w-ij is not empty, and starts with a terminal. Let's define an auxiliary function of the k-the prefix of a word (a-1,…,a-n):

prefix : Alp^{c*} x {1, 2, …} ⟼ Alp^{c*}
prefix(w, k) =
  w = () ➔ ()
  let
    (a-1,…,a-n) = w
  n ≤ k ➔ w
  true ➔ (a-1,…,a-k)

For a positive integer k, a strongly prefixed grammar with equations (7.2-1) is said to be a LL(k) grammar[20], if for every index 1≤ i ≤ n, any two different metawords in the i-th equation, w-ij and w-ip, have different k-th prefixes, i.e., prefix(w-ij, k) ≠ prefix(w-ip, k). Note that metawords of different equations do not need to satisfy this condition.

In a LL(k) grammar, given a word w to be parsed, and a non-terminal Syn-i that determines the category of this word, we need to look ahead not more than k first characters of w to identify the grammatical clause to be used in parsing w. This property of LL(k) grammars allows to build for them relatively simple deterministic parsers.

As is easy to check, our abstract-syntax grammar is LL(k) for some k, since all green prefixes of clauses are different to each other.

[20] The original concept of a LL(k) grammar is not restricted to strictly prefixed grammar, but in that case the definition is a little more complicated, and requires the introduction of some additional concepts. On the other hand, the restriction to strictly prefixed grammar is not harmful for our model, since our abstract-syntax and concrete-syntax grammars will be strictly prefixed anyway.

3 AN INTUITIVE INTRODUCTION TO DENOTATIONAL MODELS

3.1 How did it happen?

Mathematicians building mathematical models of programming languages were usually assuming (as in mathematical logic) that a programming language should be described by three mathematical entities:

1. Den — *denotations,* which in our model constitute a many-sorted algebra (Sec. 2.12),
2. Syn — *syntax,* which in our model is an algebra similar to the former (has the same signature),
3. Sem : Syn ↦ Den — *semantics,* that associates denotations to syntactic elements, and in our model is a homomorphism between two mentioned algebras.

Intuitively speaking, a denotational semantics describes the meaning of every complex syntactic object as a composition of the meanings of its components. This property of semantics — called *compositionality* — allows for the description of complex objects by means of so-called *structural induction.*

It should be mentioned in this place that denotational (compositional) models of semantics — which for mathematicians have always been an obvious choice — have not been used in the first formal models of programming languages. Similarly to the prototypes of sewing machines that were mechanical arms repeated the movements of a tailor, and to the first steamboat engine droving oars, the early formal definitions of programming languages were mathematical descriptions of virtual computers executing programs[21].

Such model of semantics, called later *operational semantics*, were abandoned after a few years of experiments because descriptions of virtual machines were not less complex than the codes of a compilers, and still they weren't descriptions of "real" machines[22].

However, the road to denotational semantics wasn't simple either. As was already mentioned, early denotational models of programming languages were characterized by great mathematical complexity. Technically it was the consequence of the assumption that two following mechanisms were indispensable in high-level programming languages:

1. the jump instruction **goto** that transfers program execution from one line of code to another one; this mechanism was available in practically all programming languages in the years 1960/70, and was inherited from low-level languages, where it was the only tool for building logical structures of programs,
2. procedures that may take themselves as parameters; this construction was present in Algol 60 (see [7]) considered by academic community of 1960. as an indisputable standard.

[21] First metalanguage used to write such semantics in the 1970. was developed in IBM laboratory Vienna and was called Vienna Definition Language (VDL). Later some members of the IBM team have created a lab on the Danish Technical University in Lyngby with the aim of writing a denotational semantics in a metalanguage called Vienna Development Method (VDM) [15]. This language was used, among other applications, to describe the semantics of two programming languages — Ada and Chill. In the case of the former, that was expected to become a universal programming language of all times, the process of writing its semantics resulted in repairing many inaccuracies of the language, and in developing first Ada compiler. Unfortunately, both Chill and Ada were excessively complex, and hence have never became commonly used.

[22] To be precise this remark is true for sequential programming only, i.e. without concurrency. An operational semantics for concurrent programs was developed by Plotkin [81].

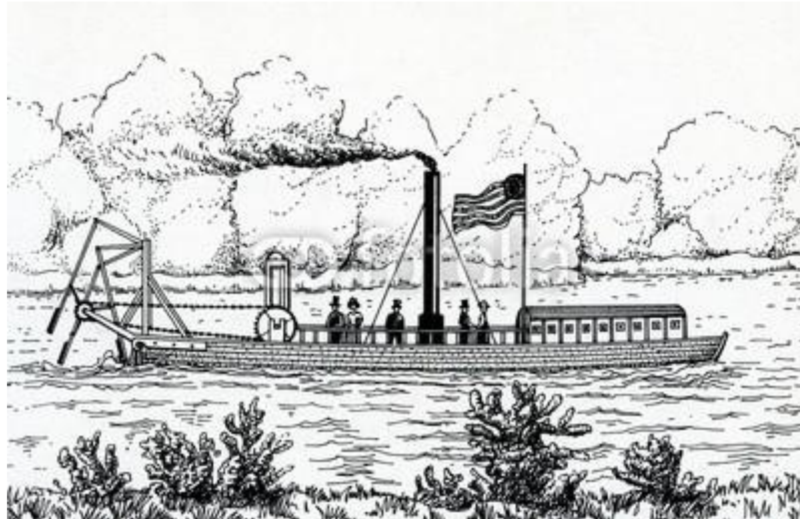


Fig. 3.1-1 Steamboat moving oars

The requirement of having **goto**'s has led to a technically rather complex model of continuations[23]. That semantics was not only technically complex but above all quite far from programmers' intuition. Independently, at the turn of the 1960-ties to 1970-ties, IT professionals began to be aware of a risk imposed by **goto** instruction (see [49]). Programs with goto's were difficult to understand, and therefore not always behave as expected. As a consequence **goto**'s were abandoned in favor of structural programming mechanisms such as **if-the-else, while-do-od** and similar.

The continuation model, although technically complex, was based on a traditional mathematics. This can't be said about the model of procedures that take themselves as parameters. Notice that in this case we do not talk about recursive procedures that call themselves in their bodies — such a mechanism is described in this book by fixed-point equations — but about constructions of the type f.f, where a function takes itself as an argument. Such functions were not known to mathematicians, because they can't be described on the ground of classical set theory, let alone that mathematicians never needed such functions.

In Algol 60 the construction f.f was implemented in such a way, that a procedure f was receiving as a parameter not exactly itself, but a copy of its own code, which was inserted into its body during compilation. Such an operation was called *copy rule*. Mathematicians of the decade of 1960. were fascinated by this construction because it was challenging the existing concept of a function. As a consequence, the theory of *reflexive domains* was created by Dana Scott and Christopher Strachey [84] and was later described in detail by J.E. Stoy in a monograph [83][24]. Although some mathematicians were investigating reflexive domains, for software engineers this theory was even more difficult, and less intuitive then continuations. Pretty soon it turned out also that the self-applicability of procedures was even more error-prone than the use of **goto**'s. Consequently, in later programming languages, self-applicable procedures were abandoned. Unfortunately, some researchers decided that denotational semantics should be abandoned as well.

In the denotational model discussed in this book we use neither continuations nor reflexive domains. In our model the denotations of instructions are state-to-state functions where a state "carriers" everything that a program needs to be executed: data, types, procedures, classes etc. Simplifying a little a state is a function

---

[23] First author who introduced that concept — although under a different name of *tail functions* — was Antoni Mazurkiewicz [71]. Under the name of continuations it was introduced in [84]**Błąd! Nie można odnaleźć źródła odwołania.** and later and popularized in [83].

[24] To our colleagues mathematicians we may explain that the idea of reflexive domains was in fact a "hidden realization" of copy rule. The authors of this model used the fact that functions definable by programs are computable, hence can be "numbered" with natural numbers — each function f may be given a unique number n(f). In this model f(f) meant f(n(f)) which can be modelled on the ground of classical set theory. That was in fact a mathematical application of copy rule since n(f) may be regarded as the code of procedure f.

that maps identifiers into these mathematical items. The concept of a state is a natural generalization of a concept of a *valuation* known by mathematicians since the pioneering works of Alfred Tarski [85]. Tarski defined the meanings of expressions as functions mapping valuations of variables

val : Valuation = {x, y, z} → Value

into values. E.g., the meaning of an expression

2x+4y

was a function

F.[2x+4y] : Valuation → Number

such that

F.[2x+4y].val = 2*val.x + 4*val.y

From there only one step to an observation that the meaning of an instruction

x := 2x + 4y

is such a transformation of valuations where the value of x in the new valuation is the value of the expression 2x+4y in the former. This idea was applied in [18], published in 1971, where Andrzej Blikle described a prototype of a denotational semantics of a very simple programming language.

In turn, the inspiration to abandon the model of reflexive domains came to me from the book of Michael Gordon [59], where the author treats Scott's reflexive domains as "usual sets" with the following commentary on page 29:

We shall not discuss the mathematics involved in Scott's theory at all; our approach to recursive equations[25] is similar to an engineering approach to differential equations, namely we assume they have solutions but don't bother with the mathematical justification.

Andrzej Blikle read Gordon's book in the year 1981 during a train ride from Copenhagen to Århus, where he was going to meet Peter Mosses, a strong proponent of the theory of Dana Scott. The book was, for him, a significant breakthrough since, for the first time, he was reading a semantics of a programming language with an understanding not only of its mathematics but also of its IT content. The treatment of reflexive domains as "usual sets" was a real simplification. He also had the impression that this informal treatment did not lead to any mathematical problems. Only later, he realized that Gordon was actually not dealing with self-applicable functions.

The approach of Michael Gordon, although intuitively simple, was mathematically not entirely acceptable since reflexive domains are not "usual" sets. It wasn't, therefore, clear, whether his model did not include inconsistencies.

To cope with this problem, A.Blikle and A. Tarlecki published in 1983 a paper [39], in which they constructed a denotational model of a programming language, where the domains of denotations are sets, and the denotations of instructions are state-to-state transformations. This approach stimulated in 1980-ties the creation of a metalanguage **MetaSoft** [29] in the Institute of Computer Science of the Polish Academy of Sciences. And this is the approach that we shall discuss and further developed in this book.

3.2 From denotations to syntax

All early works on the semantics of programming languages were devoted to building semantics for existing languages. This fact has led to a tacit assumption that in designing a language, the syntax should come first into the play. Of course, there is a certain logic in this way of thinking, since how can we build a model for

---

[25] M. Gordon is talking here about recursive domain-equations, which, in some case of non-continuous domain operators, lead to D. Scott's reflexive domains.

something that does not yet exist? After all, astronomers were describing the mechanics of celestial bodies when the Sun and the planet were already there.

This way of thinking has, however, a particular vulnerability, since computer science cannot be compared to astronomy, physics, or biology, where we describe the world around us. Building a programming language is an engineering task, such as constructing a bridge or an airplane. Would any engineer ever think of first constructing a bridge basing on common sense and only then making all necessary calculations? Such a bridge would certainly collapse.

In our approach, we reverse the traditional order where one first builds a syntax, and only later defines its meaning. We will build a language starting from an algebra of detonation from which syntax will be derived in such a way that a denotational semantics exists. This construction was sketched in Sec. 2.13.

An experimental programming language developed in this book is called **Lingua**. This Italian name has been suggested by Andrzej Blikle to commemorate the circumstances under which — working as a scholar of Italian government from October to December 1969 — he wrote his habilitation thesis later published in Dissertationes Mathematicae [18]. During three months in the Istituto di Elaborazione dell'Informazione in Pisa he described a denotational semantics of a very simple programming language, although he didn't call his semantics in this way. The name "denotational semantics" was used for the first time in a joint work by D. Scott and Ch. Strachey [84]. Only eighteen years later, in the year 1987, Andrzej Blikle described (in [30]) the idea of deriving syntax from detonations.

3.3 Why we need denotational models of programming languages?

A denotational model of a programming language serves as a starting point for the realization of three tasks:

1. building an implementation of the language, i.e., its interpreter or compiler,
2. creating rules of building correct specified programs in this language,
3. writing a user manual.

When designing our language in this book, we shall observe two fundamental (although not quite formal) principles:

> ***First Principle of Simplicity***
>
> *A programming language should be as simple to understand and easy to use as possible without harming its functionality, mathematical clarity, and completeness of its description.*
>
> ***Second Principle of Simplicity***
>
> *The same applies to the manual of the language and to the rules of building correct programs.*

These principles shall be fulfilled by:

1. making the syntax of the language as close as possible to the language of "usual" mathematics, e.g., whenever it is common, we allow infix notation and the omission of "unnecessary" parentheses,
2. making the semantics of the language easy to understand by the user rather than convenient for the implementor; for the latter, an equivalent implementation-oriented model may be written.
3. making the structure of the language (i.e., program constructors) leading to possibly simple rules of constructing correct programs (Sec. 8 and Sec. 9),

Particular attention should be given to point 3. because the simplicity of the rules of building correct programs leads to a better understanding of programs by programmers. This fact was realized already in the decade of 1970. and has led to the elimination of **goto** instructions. This decision led to a significant simplifica-

tion of program structures, which increased their reliability. On the other hand, it did not limit the functionality of programming languages.

Following point 3, we will sometimes — as typical in mathematics — "forget" about the difference between syntax and denotations. E.g., we will talk about the value of an arithmetic expression x + y, rather than about the value generated by its denotation. We will say that the instruction x:=y+1 modifies the value of x, instead of saying that the denotation of this instruction modifies a memory state at variable x, etc. Of course, at a formal level, we shall precisely distinguish syntax from denotations.

3.4 Five steps to a denotational model

Building up **Lingua,** we refer to an algebraic model described in Sec. 2.11 to Sec. 2.16. It corresponds to the diagram of three algebras shown in Fig. 3.4-1. We build it in such a way that the equation:

A2D = A2C • C2D

is satisfied, which guarantees the existence of a denotational semantics of our language.

The construction of a denotational model begins with a description of an algebra of detonation **AlgDen**. Then from the signature of **AlgDen** we derive an *algebra of abstract syntax* **AlgAbsSyn**, and, precisely speaking a context-free grammar that describes this algebra. The first of these steps is creative since it comprises all the significant decisions about a future language. In turn, the second step can be performed algorithmically.

Since abstract syntax is usually not convenient for programmers, we build an *algebra of concrete syntax* **AlgConSyn**. In typical situations, we do it by replacing prefix notation by infix notation and introducing more intuitive names of constructors. In our approach the corresponding abstract-to-concrete homomorphism A2C will be an adequate homomorphism, which guarantees the existence of a unique homomorphism:

C2D : **AlgConSyn** ↦ **AlgDen**

(*concrete semantics*), which is the semantics of concrete syntax. In this way, we create the main components of our denotational model.

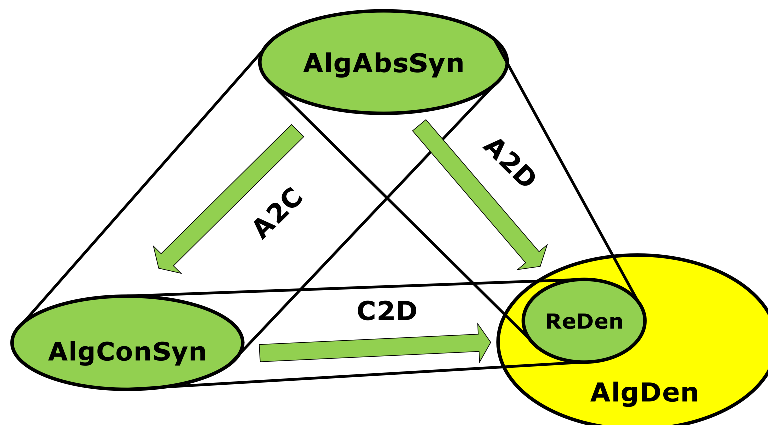


Fig. 3.4-1 Basic algebraic model of a programming language

The step from abstract syntax to concrete syntax is creative — although rather simple. For instance, instead of writing +(a, b) we write (a + b) and instead of writing

if.(greater.(x, 0), assign.(x, plus.(x, 1)), assign.(x, minus.(x. 1)))

we write

if x>0 **then** x:=x+1 **else** x:=x-1 **fi**

The next step in building a user-friendly syntax consist in introducing so called *colloquialisms*. For instance instead of writing

(a+(b+(c*d))

we shall write

a + b + c*d

assuming that multiplication binds stronger than addition, and that "the remaining" parentheses are added from left to right. The introduction of colloquialisms into concrete syntax leads to an *algebra of colloquial syntax* **ColSyn** (Fig. 3.4-2), which most frequently has a different signature than concrete syntax, and therefore can't be a homomorphic image of it. However, we make sur that there exists an implementable *restoring transformation*

RES : AlgColSyn ⟼ AlgConSyn

that transforms colloquial syntax back to the concrete one, e.g., by adding the missing parentheses.

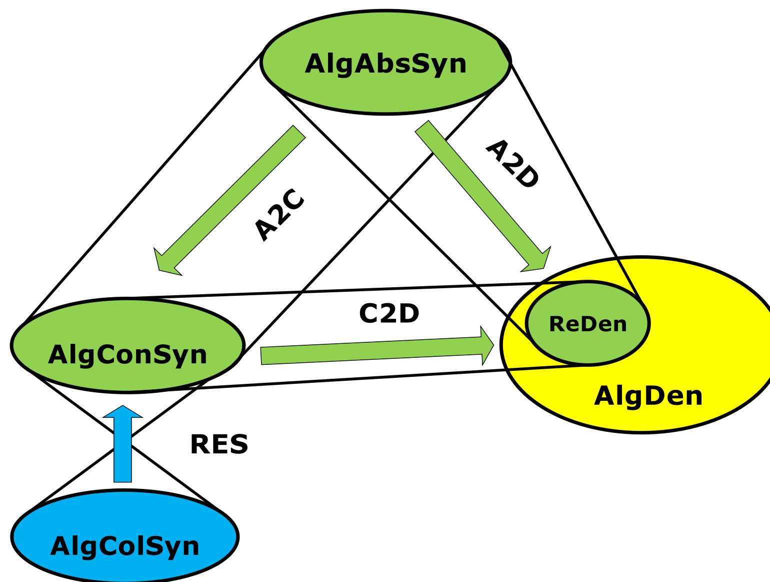


Fig. 3.4-2 An algebraic model of a language with colloquial syntax

In a programmer's manual, a language with colloquialisms is described by a grammar of concrete syntax with additional clauses and a restoring transformation (Sec. 7.4). For instance, we explain that in writing arithmetic expressions, we can skip parentheses while maintaining the priority of multiplication and division over addition and subtraction.

In such a case, an implementor receives a standard denotational model of a language plus a formal definition (algorithm) of restoring transformation. The execution of a program consists then of two steps:

1. a pre-treatment of the source code by a restoring transformation,
2. an interpretation or compilation of concrete-syntax code.

Summing up our considerations, the construction of a denotational model of a programming language correct-program constructors proceeds in five steps:

1. In the first step, we build an algebra of detonations **AlgDen** that includes the denotations of the future syntax as well as their constructors. In that step, significant decisions are taken about the functionality of the language. A language designer must specify the repertoire of constructors of **AlgDen** in such a way that the corresponding (unique) reachable subalgebra contains all the elements that we want to access through syntax. This will be illustrated and explained in Sec. 6. In the earlier Sec. 3.5 and Sec.

5 we build technical fundaments for the algebra of denotations — data- and type-oriented algebras, objects, classes and states.

2. The signature of algebra **AlgDen** uniquely determines the algebra of abstract syntax **AlgAbsSyn** and the corresponding homomorphism (abstract semantics) A2D. Formally this step (Sec. 7.2) leads from the signature of **AlgDen** to an equational grammar of **AlgAbsSyn**, and can be performed algorithmically.
3. Since abstract syntax is not user-friendly, we transform it (Sec. 7.3) in a homomorphic way to a concrete syntax **AlgConSyn**, which is closer to programmers' syntax. We make sure that this homomorphism is adequate which guarantees the existence a denotational semantics (a homomorphism C2D : **AlgConSyn** ↦ **Den**.
4. In the fourth step, we introduce colloquialisms (Sec. 7.4) — which make our language even more user-friendly — and describe the restoring transformation. This step is creative again. The grammar of colloquial syntax emerges from the grammar of concrete syntax by adding to it some new grammatical clauses.
5. In the last step we build tools for the construction of correct programs (Sec. 9). In our opinion this step should be regarded as an inherent phase in designing a programming language. It should be the responsibility of a language designer to choose such programming mechanisms which make the corresponding constructors of correct programs sufficiently easy to use.

In the end let us reemphasize that **Lingua** is not regarded as a prototype of a stand-alone applicative programming language, but only as an example of a language with denotational semantics.

3.5 Six steps to the algebra of denotations

On the ground of our model we suggest a certain systematic way of getting to an algebra of denotations of a future object-oriented language.

1. In the first step we decide about the categories of data that we want to have in the language and a corresponding set of constructors. Typically, we may start by defining some simple data, e.g., numbers or texts, and some structured data, e.g., lists or arrays.
2. The mentioned categories of data correspond to *data types.* Data types correspond to sets of data, but they are not such sets. They are independent mathematical beings that only describe such sets. This solution allows us to assume that whenever we build a (new) data, we "simultaneously" build its type.
3. To formalize the described mechanism we introduce *typed data* that are pairs consisting of data and their types. Each constructor of typed data, given a tuple of arguments ((dat-1, typ-1),…,(dat-n, typ-n)) builds a new typed data by applying a data constructor to (dat-1,…,dat-n) and a corresponding type constructor to (typ-1,…,typ-n).
4. Typed data constitute one of two categories of *values*. The second category are *objects* that are pairs consisting of an *objecton* and its type. Objectons are typed memory structures and their types are the names (identifiers) of corresponding classes. Classes, in turn, are structure that carry types, methods and objectons.
5. Since denotations in our model are (with some exceptions) functions on *states*, in the last but one step we define states. The latter carry classes, values assigned to variables (identifiers) via references and some other elements of a technical character. References represent typed memory addresses carrying (sort of) predicates called *yokes.* Yokes describe type-independent properties of values.
6. In the last step we define an *algebra of denotations*, i.e., its carriers and constructors. We will have two major categories of denotations: *applicative denotations* — the denotations of expressions, and *imperative denotations* — the denotations of declarations and instructions.

Although in steps 1., 2. and 3. we might talk about building algebras, we do not formalize this fact, since in building our model we refer merely to their elements and constructors. In turn, in the case of denotations, we

define a corresponding algebra explicitly, since later on we shall derive from this algebra our algebras of syntaxes.

3.6 Lingua as a strongly-typed language

In a manual of SQL ([52] p. 786), we can read the following sentence[26]:

"If we do not provide (…) correct values to functions as their arguments, we should not expect consistent results."

Contrary to this philosophy, **Lingua** will be constructed in such a way that whenever a program provides unexpected values to a function, this function will generate an error message and/or initiate a recovery action. To achieve this goal, we equip **Lingua** with a *typing discipline* partly announced in Sec. 3.5. In **Lingua** "incorrect values" means "values of not acceptable types". Types will be used in the descriptions of the following mechanisms:

1. the declarations of variables,
2. the declarations of user-defined types,
3. the evaluation of expressions,
4. the execution of assignment instructions,
5. passing arguments to operations on values,
6. passing actual parameters to all three categories of procedures — imperative, functional and object constructors,
7. returning reference parameters at the end of imperative-procedure calls,
8. returning values of functional procedures,
9. defining the types of formal parameters of all categories of procedures.

[26] Andrzej Blikle's translations from a Polish edition [52].

4 DATA, TYPES, VALUES AND YOKES

4.1 Data

The first step of designing a programming language in our framework consists in defining *data* and their constructors, i.e., an *algebra of data*. It is to be emphasized in this place that in our model, we will have two categories of algebras:

- *first-class algebras* — algebras of denotations and the corresponding algebras of syntaxes,
- *second-class algebras* — algebras of data, data types, typed data and yokes.

In the first case we have to make sure that algebras of denotations have nonempty reachable subalgebras, since that is necessary for the algebras of syntaxes (reachable by definition), to be not empty. For these algebras we introduce metavariables (names) **AlgDen** or **AlgAbsSyn** and we define homomorphism between them, e.g.:

A2D : **AlgAbsSyn** ⟼ **AlgDen**

The situation with second-class algebras is different. We do not introduce metavariables for them, and we do not care about their reachable parts. However, we still refer to them as "algebras" to express that they represent collections of some elements and some operations on them.

Proceeding to our algebra of data we recall and reemphasise that in our book we are not building a real programming language, but only indicate how such a language might be designed. Consequently, our operations on data do not need to constitute a complete set of operations. They only offer some typical examples of such operations and their definitions.

To begin with, we assume to be given some *simple data* offered by an implementation platform. We shall not define them explicitly assuming that they are just parameters of our model. Let's assume, therefore, that we are given the following (somehow defined) domains of simple data offered by an implementation platform:

int : Integer = …
rea : Real = …
boo : Boolean = {tt, ff}
tex : Text = …

and that all these domains (except Boolean) are additionally somehow restricted by a limitation of the size of their elements, e.g.,

int : Integer = $[-2^{31}, 2^{31}-1]$

We assume further to be given a set of corresponding *prime constructors*, defined on simple data and again offered by an implementation platform, such as, e.g.,

pr-divide-in : Integer x Integer → Integer   prime division of integers[27]
pr-divide-re : Real x Real → Real   prime division of reals

In the general case we may assume that these functions are partial by which we mean that their executions may either yield no value (e.g., looping indefinitely), or return an "unwanted" value[28]. Let

[27] We assume that the result of this operation within the range of the prime integers is the integer part of the rational result of the "mathematical" division of integers.

dat : SimData = Boolean | Integer | Real | Text

be the domain of simple data, and let

ide : Identifier = …

be a set of (somehow defined) identifiers. On this ground we define a domain of *data* that are either simple or *structured*:

dat : Data = SimData | List | Array | Record
lis : List = $Data^{c*}$
arr : Array = Integer ⇒ Data
rec : Record = Identifier ⇒ Data

A list is a finite, possibly empty, tuple of arbitrary data. Arrays and records are mappings, i.e., finite functions. Arrays are one-dimensional, but since their elements can be arrays themselves, our model includes arrays of arbitrary dimensions. Identifiers which are in the domain of a record will be called the *record attributes*.

All domains of data, except SimData, will be referred to as *data sorts,* e.g., *integer sort*, *text sort, array sort,* etc. At this stage, lists and arrays are not-homogeneous, i.e., may include elements of different sorts, and may be arbitrarily large. Later the constructors of values, i.e. typed data (Sec. 4.3) will assure that all data generated by programs will have "appropriate" structures and sizes. The technique of defining "oversized" domains whose implementable parts are later appropriately "truncated" is typical for denotational models since it leads to simple domain equations. We will frequently use it in the sequel[29].

To define our data constructors we assume to be given a universal domain Error of all "future" error messages and that with every domain of data we associate a corresponding domain with errors, e.g.,

int : IntegerE = Integer | Error.

Having defined data domains, we may proceed to the definitions of data constructors. We start with their signatures and give some of their definitions a little later. Since we regard the domains and the constructors of data as a parameters of our model, their definitions should be regarded as examples only. The names of data constructors are prefixed with da- which stands for "data". Later we will have type constructors, value constructors, denotation constructors etc.

Comparison constructors

da-equal	: DataE x DataE	⟼ BooleanE	data comparison
da-less	: DataE x DataE	⟼ BooleanE	data comparison

Formally these two constructors are defied for all data. It does not mean, however, that we intend to compare lists or arrays among them or even lists with arrays. In all such cases we may assume that our constructors return error messages.

At this stage we do not introduce logical connectives and, or and not in the domain BooleanE. They will come into play only at the level of expression denotations in Sec. 6.4.2, and this decision will be explained there.

Integer number constructors

da-add-in	: IntegerE x IntegerE	⟼ IntegerE
da-subtract-in	: IntegerE x IntegerE	⟼ IntegerE

---

[28] This may happen, e.g., if the implementation platform provides an addition modulo say 10^9, where 999999999 + 1 = 0.

[29] At the level of the algebra of denotations "implementable" would mean "algebraically reachable" (cf. Sec. 2.13). As we remember, only reachable denotations are representable in syntax. However, at the level of data — and later of types and values — we do not need to care about reachability, since these elements won't have syntactic counterparts. Indeed, if we write e.g. 17.3 in a program, it is not a syntactic representation of the corresponding number, but of an expression whose fixed value is that number. This will be formalized in Sec. 6.4.2

da-multiply-in	: IntegerE x IntegerE	⟼ IntegerE	
da-divide-in	: IntegerE x IntegerE	⟼ IntegerE	

Real number constructors

da-add-re	: RealE x RealE	⟼ RealE	
da-subtract-re	: RealE x RealE	⟼ RealE	
da-multiply-re	: RealE x RealE	⟼ RealE	
da-divide-re	: RealE x RealE	⟼ RealE	

Text constructors

da-glue-te	: TextE x TextE	⟼ TextE	

List constructors

da-empty-li	:	⟼ ListE	
da-put-to-li	: DataE x ListE	⟼ ListE	
da-head-li	: ListE	⟼ DataE	
da-tail-li	: ListE	⟼ ListE	

Array constructors

da-empty-ar	:	⟼ ArrayE	create an empty array
da-put-to-ar	: ArrayE x DataE	⟼ ArrayE	add an element with a "next" index
da-change-in-ar	: ArrayE x IntegerE x DataE	⟼ ArrayE	replace an element of an array
da-get-from-ar	: ArrayE x IntegerE	⟼ DataE	

Record constructors

da-create-rc	: Identifier x DataE	⟼ RecordE	
da-put-to-rc	: DataE x RecordE x Identifier	⟼ RecordE	
da-get-from-rc	: RecordE x Identifier	⟼ DataE	
da-change-in-rc	: RecordE x Identifier x DataE	⟼ RecordE	replace an element of a record

Notice that among our constructors, we do not have constructors of identifiers. We return to them at the level of value-expression denotations in Sec. 6.4.2.

In order to define simple-data constructors, we assume to be given some *prime constructors* which we may think of as provided by an *implementation platform*.

It is a well-known fact that for some arguments prime constructors return either a wrong answer or no answer at all. E.g., we can't divide a number by zero, or can't add two numbers if their sum would be too large for the current implementation. In all such cases our data constructors should not be performed in a "standard way", but instead an error message should be generated. The same concerns the constructors of structured data. For instance, we may wish to set a limit to the volume of an array.

To systematically incorporate this mechanism, into our model, with every data constructor we associate an auxiliary function called a *trust test.* E.g. with real division we associate a trust test:

trust-da-divide-re : RealE x RealE ⟼ Error | {'OK'}

such that whenever the primary constructor pr-divide-re does not return a correct arithmetical result, the trust test yields appropriate error message, and otherwise it generates 'OK'. For instance we may set:

```
trust-da-divide-re.(rea-1, rea-2) =
    rea-i : Error                      ➔ rea-i for i = 1, 2
    rea-2 = 0                          ➔ 'division by zero not allowed'
    pr-divide-re.(rea-1, rea-2) > max-int  ➔ 'overflow'
    pr-divide-re.(rea-1, rea-2) < min-int  ➔ 'underflow'
    true                               ➔ 'OK'
```

where pr-divide-re is the prime division, and max-rea and min-rea denote the maximal/minimal real acceptable in a current implementation. Of course, the predicates

pr-divide-re.(rea-1, rea-2) > max-rea
pr-divide-re.(rea-1, rea-2) < min-rea

must be "somehow" implemented.

We assume that all our trust tests will be transparent for errors (Sec. 2.9). In this book we shall not define trust tests explicitly assuming that they constitute yet another category of parameters of our model.

Given a trust test for the division of reals, the definition of the corresponding data constructor will be the following:

```
da-divide-re.(rea-1, rea-2) =
   trust-divide-re.(rea-1, rea-2) : Error   ➔ trust-divide-re.(rea-1, rea-2)
   true                                     ➔ pr-divide-re.(rea-1, rea-2)
```

Another example of a trust test is associated with a constructor that adds an element to a list:

```
trust-add-to-li : DataE x ListE ⟼ Error | {'OK'}
trust-add-to-li.(dat, lis) =
   dat   : Error                  ➔ dat
   lis   : Error                  ➔ lis
   size.(push.(dat, lis)) > max   ➔ 'overflow'
   true                           ➔ 'OK'
```

where push is a "mathematical" functions defined in Sec. 2.2, size is a function that somehow computes the memory size necessary to "fit" the list, and max is a parameter of our model[30]. The definition of the corresponding operation on lists will be the following:

```
da-add-to-li : DataE x ListE   ⟼ Error | {'OK'}

da-add-to-li.(dat, lis) =
   truth-da-cons.(dat, lis) : Error   ➔ truth-da-cons.(dat, lis)
   true                               ➔ push.(dat, lis)
```

The definitions of the remaining data constructors are analogous and we assume them to be parameters of our model.

Note that our algebra of data is not reachable, since we have not defined any zero-argument data constructors. As was already mentioned, we shall only care about the reachability of algebras at the level of denotations.

It is to be emphasized at the end that we use trust tests only at the level of data. We "sort them out" from the definitions of data constructors just to emphasize that their identifications constitute an essential step in designing a programming language. In the book's sequel, practically all defined constructors will perform some adequacy checks of their arguments, but we shall not define these checks as separate trust tests.

4.2 The types of data

Having defined data and their constructors we may proceed to the types of data otherwise called *data types*. Data types describe "internal structures" of data, and will become components of values. Formally data types are defined as words, tuples, mappings or their combinations. The categories of data types reflect possible structures of data:

---

[30] In this definition, and in all definitions in the sequel, we assume that whenever an error appears in a computation, this computation is aborted and the error is signalized, i.e., is returned as a terminal result. In our case if both **dat** and **lis** are errors, then dat-error is signalized. An alternative to this solution could be that we search for, and signalize, all errors. Since such a solution would significantly lengthen our definitions, and made them less readable, we gave it up. After all **Lingua** is only an example.

typ : DatTyp =
  {'integer', 'real', 'boolean', 'text'} | simple types
  {'L'} x DatTyp | list types
  {'A'} x DatTyp | array types
  {'R'} x (Identifier ⇒ DatTyp) record types

Types of simple data are one-element tuples of words. Symbols 'L', 'A' and 'R' are called *type initials* and indicate the *sorts* of structural types. E.g. ('A', 'integer') is the type of arrays of integers, and ('L', ('A', 'real')) is the type of lists whose elements are arrays of reals.

In the case of a list-type ('L ', typ) we say that typ is the *inner type* of the list type and similarly for array-types. The elements of the domain

tyr : TypRec = Identifier ⇒ DatTyp

are called *type records,* e.g.:

employee-type =
  ['ch-name' / 'text',
  'fa-name' / 'text',
  'award-years' / ('A', 'integer'),
  'salary' / 'integer',
  'commission' / 'integer']

Type records, i.e., records of types, should not be confused with record types, that are types of records. A record type consist of a record initial 'R' and a type record. Other examples of data types may be:

('L', ('R', [name/'text', age/'integer'])) a type of lists of records
('A', ('L', ('R', [name/'text', age/'integer']))) a type of arrays of lists of records

The definition of the domains of types anticipates the principle that all elements of a list or of an array must have a common type.

Notice that an array type does not specify the number of array elements. The introduction of arrays with a fixed number of elements will be possible with the use of yokes (see Sec. 4.4).

To associate data with data types, we define with each type a set of data called the *clan* of this type. Formally, we define a function CLAN-ty that with each type assigns its clan:

CLAN-ty : DatTyp ↦ Sub.Data

This function is defined by structural induction

CLAN-ty.'boolean' = Boolean
CLAN-ty.'integer' = Integer
CLAN-ty.'real' = Real
CLAN-ty.'text' = Text
CLAN-ty.('L', typ) = $(\text{CLAN-ty.typ})^{c*}$
CLAN-ty.('A', typ) = Integer ⇒ CLAN-ty.typ
CLAN-ty.('R', [ide-1/typ-1,…, ide-n/typ-n]) =
  { [ide-1/dat-1,…, ide-n/dat-n] | dat-i : CLAN-ty.typ-i for i = 1;n }

An important fact to be signalised in this place is that the union of the clans of all types not does not exhaust the domain Data. In other words, there exist data which have no types. An example of such a data may be a

non-homogeneous list such as, e.g., (123, 'abc', tt). As we will see in the sequel, non-homogeneous data will not "happen" in our programs. In this way, by introducing types, we restrict the set of reachable data[31].

It is also worth noticing that clans of different types need not be disjoint. E.g. the clans of types ('A', 'integer') and ('R, [ ]) both include empty functions.

For technical reasons we introduce an auxiliary function of a *sort of a type*:

sort-t : DatTyp ⟼ {'boolean', 'integer', 'real', 'text', 'L', 'A', 'R'}

sort-t.typ =
  typ = 'boolean' ➔ 'boolean'
  typ = 'integer' ➔ 'integer'
  typ = 'real' ➔ 'real'
  typ = 'text' ➔ 'text'
  typ : {'L'} x DatTyp ➔ 'L'
  typ : {'A'} x DatTyp ➔ 'A'
  typ : {'R'} x (Identifier ⇒ DatTyp) ➔ 'R'

To define constructors of data types we introduce a domain that includes data types and errors.

typ : DatTypE = DatTyp | Error

Now, for every data constructor da-co we define a data-type constructor ty-co that builds the type of the data built by da-co.

Comparison constructors

ty-equal : DatTypE x DatTypE ⟼ DatTypE
ty-less : DatTypE x DatTypE ⟼ DatTypE

Arithmetic constructors for integers

ty-add-in : DatTypE x DatTypE ⟼ DatTypE
ty-subtract-in : DatTypE x DatTypE ⟼ DatTypE
ty-multiply-in : DatTypE x DatTypE ⟼ DatTypE
ty-divide-in : DatTypE x DatTypE ⟼ DatTypE

Arithmetic constructors for reals

ty-add-re : DatTypE x DatTypE ⟼ DatTypE
ty-subtract-re : DatTypE x DatTypE ⟼ DatTypE
ty-multiply-re : DatTypE x DatTypE ⟼ DatTypE
ty-divide-re : DatTypE x DatTypE ⟼ DatTypE

Text constructors

ty-glue-li : DatTypE x DatTypE ⟼ DatTypE

List constructors

ty-empty-li : DatTypE ⟼ DatTypE
ty-put-to-li : DatTypE x DatTypE ⟼ DatTypE
ty-head-li : DatTypE ⟼ DatTypE
ty-tail-li : DatTypE ⟼ DatTypE

Array constructors

ty-create-ar : DatTypE ⟼ DatTypE

[31] In this place the word "reachable" does not mean "reachable algebraically" in the sense described in Sec. 2.13. It only means that such a data may appear as a component of a value (see Sec. 4.4) generated by an expression evaluated in a program.

ty-put-to-ar : DatTypE x DatTypE ⟼ DatTypE
ty-change-in-ar : DatTypE x DatTypE x DatTypE ⟼ DatTypE
ty-get-from-ar : DatTypE x DatTypE ⟼ DatTypE

Record constructors

ty-create-rc : Identifier x DatTypE ⟼ DatTypE
ty-put-to-rc : DatTypE x DatTypE x Identifier ⟼ DatTypE
ty-get-from-rc : DatTypE x Identifier ⟼ DatTypE
ty-change-in-rc : DatTypE x Identifier x DatTypE ⟼ DatTypE

Below show a few examples of the definitions of these constructors:

ty-equal.(typ-1, typ-2) =
typ-i : Error ➔ typ-i for i = 1,2
typ-1 ≠ typ-2 ➔ 'types of compared values must coincide'
not comparable.typ-1 ➔ 'not comparable'
true ➔ 'boolean'

Here we have used a metapredicate comparable to indicate the fact that data of some types may be not comparable. E.g. we may assume that simple data are comparable, but structured data are not. Of course other solutions are possible as well.

ty-divide-in.(typ-1, typ-2) =
typ-i : Error ➔ typ-i for i = 1,2
typ-i ≠ 'integer' ➔ 'integer expected' for i = 1,2
true ➔ 'integer'

ty-empty-li.typ =
typ : Error ➔ typ
true ➔ ('L', typ)

ty-put-to-li.(typ-e, typ-l) = cons typ-e on list typ-l
typ-i : Error ➔ typ-i for i = e,l
sort-t.typ-l ≠ 'L' ➔ 'list expected'
let
('L', typ) = typ-l
typ-e ≠ typ ➔ 'conflict of types'
true ➔ typ-l

ty-empty-ar.typ =
typ : Error ➔ typ
true ➔ ('A', typ)

ty-put-to-ar.(typ-e, typ-a) = put typ-e to array typ-a
typ-i : Error ➔ typ-i for i = a,e
sort-t.typ-a ≠ 'A' ➔ 'array expected'
let
('A', typ) = typ-a
typ ≠ typ-e ➔ 'conflict of types'
true ➔ typ-a

ty-create-rc.(ide, typ) =
typ : Error ➔ typ
true ➔ ('R', [ide/typ])

ty-put-to-rc.(typ-e, typ-r, ide) = put typ-e to record typ-r on attribute ide

typ-i : Error ➔ typ-i
sort-t.typ-r ≠ 'R' ➔ 'record expected'
typ-r.ide = ! ➔ 'attribute already exist'
true ➔ ('R', typ-r[ide/typ-e])

ty-change-in-rc.(typ-r, ide, typ-e) = check if new type coincides with the former
typ-i : Error ➔ typ-i for i = r, e
sort-t.typ-r ≠ 'R' ➔ 'record expected'
let
('R', typ-rb) = typ-r -rt for „record type"
typ-rb.ide = ? ➔ 'no such attribute'
let
typ-at = typ-rb.ide -at for „attribute type"
typ-e ≠ typ-at ➔ 'conflict of types'
true ➔ typ-r

The last constructor will be used in Sec. 4.3 in the definition of a constructor of typed data that replaces a data assigned to an attribute of a record by another data. Here we anticipate the fact that if we replace a data assigned to a record attribute, the new data must have the same type as the former one.

Type constructors will play a double role in our model:

1. they will be used in evaluating value expressions to build the type of the new value,
2. they will be used in type expressions and type declarations to build user-defined data types.

4.3 Typed data

Typed data are pairs consisting of a data and its type, and their constructors will constitute a fundament for future value-expression denotations. As was already announced, a typed data is a data and its type:

tyd : TypDat = {(dat, typ) | dat : CLAN-ty.typ} (4.3-1)

In this place we should explain why we decided to operate on typed data, rather than on data alone, despite the fact that if a data has a type then this type is unique type (Sec. 4.2)? There are at least five reasons of our decision:

1. Not all data have types.
2. We want to show explicitly how the modifications of data go "in parallel" with the modification of their types. In this way, we also suggest a specific solution for **Lingua** implementation.
3. Whenever a typed data becomes an argument of an operation, or is to be assigned to a variable or to a formal parameter of a procedure, we have to check the coincidence of the type of this data with an expected type of an argument, a variable or a parameter respectively. In all such cases having an explicit type of a data is just handy.
4. As we will see in Sec. 4.4, typed data will constitute just one category of values, whereas the another category will be constituted by objects consisting of an objecton and its type. In this case one objecton may be associated with many different types.
5. In Sec. 5.4.2 we will introduce a covering relation between types such that wherever a value of typ1 is expected, we can use a value of typ2, provided that typ1 covers typ2.

A typed data that carries a simple data is called *simple typed-data* and analogously are understood *structural typed-data*. The constructors of typed data will "call" the corresponding constructors of data and of types. To describe this mechanism we expand the earlier introduced function sort-t (Sec. 0) onto typed data:

sort-td.(dat, typ) = sort-t.typ

Note in this place that although data were split onto several domain, we "glue" typed data into one domain. We can do so without losing a typing discipline, since data are coupled with types, and therefore the constructors of typed data may signalize errors whenever they receiving arguments of inappropriate types. As we will see in Sec. 6.4.2, this solution also leads to one carrier of value expressions denotations instead of many carriers such as, e.g., boolean expression denotations, integer expression denotations, etc. This decision simplifies our model.

Now, we proceed to the constructors of typed data. For each data constructor da-co we define a typed data constructor td-co, that "calls" (with one exception, for empty lists) the corresponding da-co and ty-co. Note that all our constructors are total functions which is possible due to the fact that TypDatE includes abstract errors.

Comparison constructors

td-equal	: TypDatE x TypDatE	↦ TypDatE
td-less	: TypDatE x TypDatE	↦ TypDatE

Arithmetic constructors for integers

td-add-in	: TypDatE x TypDatE	↦ TypDatE
td-subtract-in	: TypDatE x TypDatE	↦ TypDatE
td-multiply-in	: TypDatE x TypDatE	↦ TypDatE
td-divide-in	: TypDatE x TypDatE	↦ TypDatE

Arithmetic constructors for reals

td-add-re	: TypDatE x TypDatE	↦ TypDatE
td-subtract-re	: TypDatE x TypDatE	↦ TypDatE
td-multiply-re	: TypDatE x TypDatE	↦ TypDatE
td-divide-re	: TypDatE x TypDatE	↦ TypDatE

Text constructors

td-glue-tx	: TypDatE x TypDatE	↦ TypDatE

List constructors

td-empty-li	: DatTypE	↦ TypDatE
td-put-to-li	: TypDatE x TypDatE	↦ TypDatE
td-head-li	: TypDatE	↦ TypDatE
td-tail-li	: TypDatE	↦ TypDatE

Array constructors

td-empty-ar	: DatTypE	↦ TypDatE
td-put-to-ar	: TypDatE x TypDatE	↦ TypDatE
td-change-in-ar	: TypDatE x TypDatE x TypDatE	↦ TypDatE
td-get-from-ar	: TypDatE x TypDatE	↦ TypDatE

Record constructors

td-create-rc	: Identifier x TypDatE	↦ TypDatE
td-put-to-rc	: TypDatE x TypDatE x Identifier	↦ TypDatE
td-get-from-rc	: TypDatE x Identifier	↦ TypDatE
td-change-in-rc	: TypDatE x Identifier x TypDatE	↦ TypDatE

Let us show a few examples of the definitions of these constructors. All of them are transparent for errors (Sec. 2.9). In all cases constructors are defined according to a common scheme:

1. check if the argument typed data are not errors, and if they are not then,
2. compute the resulting type by a type constructor, and if no error is signalized then,
3. compute the resulting data by a primary data constructor, and if no error is signalized then,
4. combine the computed type and data into a typed data.

If in 1, 2 or 3 an error is signalized, then this error becomes the final result. Let us illustrate this scheme by an example of the division of integers:

```
td-divide-in : TypDatE x TypDatE ⟼ TypDatE

td-divide-in.(tyd-1, tyd-2) =
   tyd-i : Error    ➔ tyd-i                          for i = 1, 2
   let
      (dat-i, typ-i)   = tyd-i                       for i = 1, 2
      typ              = ty-divide-in.(typ-1, typ-2)
   typ : Error      ➔ typ
   let
      dat = da-divide-in.(dat-1, dat-2)
   dat : Error      ➔ dat
   true             ➔ (dat, typ)
```

In this definition, we refer to (call) two previously introduced constructors — a type constructors ty-divide-in, and a data constructor da-divide-in. First of them checks if the arguments of td-divide-in are integers, and the other is responsible for all remaining checks.

A few other examples of the constructors of typed data are shown below. Note that two of them, like the coming one, get types as arguments.

```
td-empty-li : DatTypE ⟼ TypDatE
td-empty-li.typ =
   typ : Error      ➔ typ
   let
      typ-l  = ty-empty-li.typ
   true             ➔ ((), typ-l)
```

We recall that () denotes an empty tuple, and ty-create-li.typ = (‘L’, typ). As we see, a typed list includes no data, but has a type. When we add a new typed data to such a list, its type must be typ. This rule is expressed in the following definition:

```
td-put-to-li : TypDatE x TypDatE ⟼ TypDatE
td-put-to-li.(tyd-e, tyd-l) =                                   -e – “element”, -l – “list”
   tyd-i : Error           ➔ tyd-i          for i = e, l
   let
      (dat-i, typ-i)   = tyd-i              for i = e, l
      new-typ-l        = ty-put-to-li.(typ-e, typ-l)
   new-typ-l : Error       ➔ new-typ-l
   let
      new-lis = da-put-to-li.(dat-e, dat-l)
   true                    ➔ (new-lis, typ-l)
```

When this operation is given a list that is homogeneous (all its elements are of the same type) and no error is encountered, it builds a list which is homogeneous as well. Since we ensure this property for all the constructors of list-typed data, it follows that all reachable list-typed data are homogeneous.

Note that if new-typ-l /: Error, then new-typ-l = typ-l, and therefore the type of the new list is typ-l. This definition in an unfolded (direct) form looks as follows:

```
td-put-to-li : TypDatE x TypDatE ⟼ TypDatE
td-put-to-li.(tyd-e, tyd-l) =                                   -e – “element”, -l – “list”
   tyd-i : Error           ➔ tyd-i          for i = e, l
   let
      (typ-e, dat-e)    = tyd-e
      (typ-l, dat-l)    = tyd-l
   sort.typ-l ≠ ‘L’        ➔ ‘list-type expected’
```

```
    let
        ('L' typ-le) = typ-l                                          -le – „list element"
    typ-le ≠ typ-e        ➔ 'types incompatible'
    true                  ➔ ((dat-e)@dat-l, typ-l)
```

Analogously we restrict the class of reachable array-typed data.

```
td-empty-ar : DatTypE ⟼ TypDatE
td-empty-ar.typ =
    typ : Error ➔ typ
    let
        typ-a    = ty-empty-ar.typ
        arr      = da-empty-ar.()
    true             ➔ (arr, typ-a)
```

If at the level of data we assume that

```
da-empty-ar.() = [ ],
```

The operation of putting a new element at the end of an array should guarantee that the domain of every array is of the form {1,…,n}. We set therefore

```
td-put-to-ar : TypDatE x TypDatE ⟼ TypDatE
td-put-to-ar.(tyd-a, tyd-e)   =                              put tyd-e to array tyd-a
    tyd-i : Error       ➔ tyd-i        for i = e, a
    let
        (dat-i, typ-i)  = tyd-i                              for i = e, a
        typ             = ty-put-to-ar.(typ-a, typ-e)
    typ : Error         ➔ typ
    let
        new-arr = da-put-to-ar.(dat-a, dat-e)
    true                ➔ (new-arr, typ)
```

where we assume that at the level of data we have

```
da-put-to-ar.(dat-a, dat-e) =
    dat-a = [ ]                         ➔ [1/dat-e]
    dat-a = [1/dat-1,…,n/dat-n]         ➔ [1/dat-1,…,n/dat-n, (n+1)/dat-e]
```

At the end one more definition which "inherits" a decision from the level of types:

```
td-change-in-rc : TypDatE x Identifier x TypDatE ⟼ TypDatE

td-change-in-rc.(tyd-r, ide, tyd-e) =         change in record tyd-r at attribute ide for tyd-e
    tyd-i : Error       ➔ tyd-i                for i = r, e
    let
        (dat-i, typ-i)  = tyd-i                for i = r, e
        typ             = ty-change-in-re.(typ-r, ide, typ-e)
    typ : Error         ➔ typ
    let
        new-rec = da-change-in-re.(dat-r, ide, dat-e)
    true                ➔ (new-rec, typ-r)
```

Here we assume that the corresponding data-constructor is the following:

```
da-change-in-rc.(dat-r, ide, dat-e) = dat-r[ide/dat-e]
```

The inherited decision is implicit in ty-change-in-re and concerns the fact that if we assign new data to an attribute of a record, then the new type must be identical with the previous one. Consequently the type of the record does not change.

In this place one methodological remark may be in order. In building our constructors of typed data we first built corresponding constructors of data, then of types and finally of typed data. Technically it might be simpler and shorter to define typed data constructors in one step. However, we decided to do it in a stepwise way, since it is part of the following "technological line":

1. data,
2. types,
3. typed data
4. values,
5. value expression denotations,
6. value expression syntax.

In each of these steps we concentrate on a different stage of the design of our language, i.e., on a different aspect of this language.

4.4 Yokes

As we will see in Sec. 6.7, whenever we declare a variable or a class attribute, we define the required type of its future values, i.e., of a typed data or an objects (see Sec. 4.5). The same happens when we declare formal parameters of a procedure (Sec. 6.7.4.6). This mechanism is typical for many programming languages. In some languages, however, we may declare not only the types of future values but also about some others of their properties. For instance, in SQL (Sec. 11), one may request that a column of a table has no repetitions or that two tables in a database are in a subordination relation.

To introduce such mechanisms in **Lingua**, we define a kind of[32] predicates on typed data[33], that we shall call *yokes*. At the level of syntax, they will be represented by *yoke expressions*. An example of a very simple yoke expression that describes the fact that the current value (of a variable) is greater than 10, or alternatively than the value of a variable x, is the following (we use an anticipated concrete syntax of our language described in Sec. 7.3.6):

value > 10 or **value** > x

Such a yoke will be assigned to a variable by its declaration. Another example may be

record.salary + **record**.commission < 2*x.

where + is the addition of integers. The corresponding yoke is satisfied whenever its argument is a record typed data with (at least) two attributes salary and commission, and the data assigned to these attributes are integers and satisfy the expected inequality. To be able to build such yokes we shall assume that yokes, in general, may return arbitrary typed data and not solely boolean typed data. Their domain is, therefore, the following:

yok : Yoke = TypDatE ⟼ TypDatE

Yokes that evaluate to boolean typed data or error will be called *boolean yokes*. An example of a non-boolean yoke expression is

record.salary + **record**.commission

Its denotation transforms record-typed data into integer typed data. If an argument of this yoke happens to be not a record with attributes salary and commission that carry integers, then the yoke generates an error.

A yoke is said to be *conservative,* if given an error, returns the same error. All yokes reachable in our language will be conservative. A yoke constructor is said to be *diligent,* if given conservative yokes returns conservative yokes.

---

[32] They are only "kind of predicates" rather than just "predicates", because their values are boolean values rather than just tt and ff.
[33] Technically, yokes could have been defined on arbitrary values, i.e., also on objects (Sec. 4.5), but we resign from this option for the simplicity of our model.

By the *clan of a yoke,* we mean the set of all typed data that satisfy this yoke. Formally we define a function:

CLAN-Yo : Yoke ⟼ Sub.TypDat
CLAN-Yo.yok = {tyd | yok.tyd = (tt, ‘boolean’)}

Yokes constitute an algebra with two carriers:

ide : Identifier = …
yok : Yoke = …

The carrier Yoke does not contain errors, but instead yokes may return errors as their values. We say that a typed data tyd *satisfies a yoke* yok if it belongs to the clan on that yoke.

Most yoke constructors will be derived from typed-data constructors but at the same time:

- some typed-data constructors will not generate yoke constructors,
- some yoke constructors will not be derived from typed-data constructors.

Which typed-data constructors we “bring to the level” of yokes is an engineering decision. As a matter of example we shall assume that all arithmetic constructors of typed data will have their counterparts in the algebra of yokes, whereas, in the case of arrays and records we shall make available only selection operations.

Yoke constructors may be also derived from some special data constructors that we shall not make available at the level of value expressions (Sec. 6.4.1) such as, e.g.,:

sum-in : $Integer^{c+}$ ⟼ Integer the sum of integers in the sequence
no-repet-list : $Integer^{c+}$ ⟼ Boolean no repetitions in a list
increasing-in : $Integer^{c+}$ ⟼ Boolean increasingly ordered sequence of integers

Below we list five groups of examples of yoke constructors:

1. Specific constructors not derived from constructors of typed data

yo-pass	:	⟼ Yoke	
yo-sum-li-in	:	⟼ Yoke	
yo-give-td	: TypDat	⟼ Yoke	td- stands for “typed data”

2. Constructors derived from simple-typed-data constructors (except boolean)

yo-add-in	: Yoke x Yoke	⟼ Yoke	in- stands for “integer”
yo-subtract-in	: Yoke x Yoke	⟼ Yoke	
yo-multiply-in	: Yoke x Yoke	⟼ Yoke	
yo-divide-in	: Yoke x Yoke	⟼ Yoke	
yo-add-re	: Yoke x Yoke	⟼ Yoke	re- stands for “real”
yo-subtract-re	: Yoke x Yoke	⟼ Yoke	
yo-multiply-re	: Yoke x Yoke	⟼ Yoke	
yo-divide-re	: Yoke x Yoke	⟼ Yoke	
yo-glue-tx	: Yoke x Yoke	⟼ Yoke	tx- stands for “text”

3. Constructors derived from selection constructors for structured typed data

yo-top	:	⟼ Yoke	
yo-get-from-ar	: TypDat	⟼ Yoke	ar- stands for “array”
yo-get-from-rc	: Identifier	⟼ Yoke	re- stands for “record

4. Constructors of yokes based on predicates

yo-equal-in	: Yoke x Yoke	⟼ Yoke	

```
yo-less-in           : Yoke x Yoke  ⟼ Yoke
yo-no-repet-li       :              ⟼ Yoke        li- stands for “list”
yo-increasing-li-in  :              ⟼ Yoke
```

5. Constructors of yokes based on Kleene's propositional operators

```
yo-true              :              ⟼ Yoke
yo-and               : Yoke x Yoke  ⟼ Yoke
yo-or                : Yoke x Yoke  ⟼ Yoke
yo-not               : Yoke         ⟼ Yoke

yo-all-of-li         : Yoke         ⟼ Yoke
yo-exists-in-li      : Yoke         ⟼ Yoke
yo-all-of-ar         : Yoke         ⟼ Yoke
yo-exists-in-ar      : Yoke         ⟼ Yoke
```

Our first constructor generates an identity yoke

```
yo-pass.() = pass
```

where

```
pass.tyd = tyd
```

We need this yoke for technical reason, that are explained a little later. The second constructor computes the sum of a list of integers:

```
yo-sum-li-in : ⟼ Yoke                         i.e.
yo-sum-li-in : ⟼ TypDatE ⟼ TypDatE
yo-sum-li-in.().tyd =
   tyd : Error            ➔ tyd
   sort-td.tyd ≠ ‘L’      ➔ ‘a list expected’
   let
      (dat, (‘L’, typ)) = tyd
   typ ≠ ‘integer’        ➔ ‘integers expected’
   let
      int = sum-in.dat
   int : Error            ➔ int
   true                   ➔ (int, ‘integer’)
```

An example of a yoke constructor that builds a constant-value yoke is the following

```
yo-give-td : TypDat ⟼ Yoke
yo-give-td : TypDat ⟼ TypDatE ⟼ TypDatE
yo-give-td.tyd-1.tyd-2 = tyd-1
```

This constructor, given typed data tyd-1, returns a yoke that for an arbitrary argument tyd-2 returns the typed data tyd-1. An example definition of a yoke constructor derived from a binary typed-data constructor is the following

```
yo-add-in : Yoke x Yoke ⟼ Yoke
yo-add-in : Yoke x Yoke ⟼ TypDatE ⟼ TypDatE
yo-add-in.(yok-1, yok-2).tyd = td-add-in.(yok-1.tyd, yok-2.tyd)
```

The following constructors builds a yoke which returns a selected value of an array:

```
yo-get-from-ar : TypDat  ⟼ Yoke
yo-get-from-ar : TypDat ⟼ TypDatE ⟼ TypDatE
yo-get-from-ar.ind-tyd.tyd =                        ind- stands for “index”
   tyd : Error                  ➔ tyd
   sort-t.ind-tyd ≠ ‘integer’   ➔ ‘integer expected’
   sort-t.tyd ≠ ‘A’             ➔ ‘array expected’
```

```
    let
        (dat, ('A', typ)) = tyd
    dat.ind-tyd = ?              ➔ 'index out of scope'
    true                         ➔ (dat.ind-tyd, typ)
```

The definitions of the remaining constructors of groups 2., 3. and 4. are analogous.

To explain why we need pass yoke, consider the following formula that defines the denotation of yoke expression **value** + 2:

yo-add-in.(pass, yo-in.2).tyd = td-add-in.(pass.tyd, yo-in.2.tyd) = td-add-in.(tyd, (2, 'integer'))

Note that the constructors yo-add-in must "get" to yokes as its arguments. An alternative to using pass in this example might be adding unary arithmetic constructors to our algebra, one for every integer. Since this solution would double the number of arithmetic constructors, we decided to use pass instead.

The definitions of boolean constructors of group 5. have to be defined "from scratch" since we have not defined such constructors on the level of typed data. First constructor of this group is a zero-argument constructor that returns a yoke satisfied for all values (always true):

yo-true.().tyd = (tt, 'boolean') for any tyd : TypDat

This yoke will be denoted by TT. i.e.,

TT = yo-true.()

The remaining constructors of group 5. refer to Kleene's propositional connectives (see Sec. 2.10) rather than to that of McCarthy, as it will be the case for boolean value-expressions (Sec. 6.4.1). The conjunction of yokes is defined as follows:

yo-and : Yoke x Yoke ⟼ Yoke

```
yo-and.(yok-1, yok-2).tyd =
    tyd : Error                  ➔ tyd
    let
        tyd-i = yok-i.tyd                                   for i = 1, 2
    sort-td.tyd-i ≠ 'boolean'    ➔ 'boolean expected'       for i = 1, 2
    tyd-i = (ff, 'boolean')      ➔ tyd-i                    for i = 1, 2
    tyd-i : Error                ➔ tyd-i                    for i = 1, 2
    true                         ➔ (tt, 'boolean')
```

As we see, to falsify this conjunction, it is enough that at least one of its arguments carry ff. If this is not the case, then the result is either an error or a typed-data carrying tt. Constructor yo-not is the same as in McCarthy's case, and yo-or is defined in such a way that guarantees the satisfaction of De Morgan's law, i.e.

yo-or.(yok-1, yok-2) = yo-not.(yo-and.(yo-not.yok-1, yo-not.yok-2))

The general-quantifier constructors for lists and arrays are defined in the following way (also in Kleene's spirit):

yo-all-of-li : Yoke ⟼ Yoke

```
yo-all-of-li.yok.tyd =
    tyd : Error                          ➔ tyd
    sort-td.tyd ≠ 'L'                    ➔ 'list expected'
    let
        (lis, ('L', typ)) = tyd
    lis = ()                             ➔ (tt, 'boolean')
    let
        (dat-1,…,dat-n) = lis
        tyd-i          = yok.(dat-i, typ)                      for i = 1;n
    (∃ 1 ≤ i ≤ n) tyd-i = (ff, 'boolean')➔ (ff, 'boolean')
```

```
    (∀ 1 ≤ i ≤ n) tyd-i = (tt, 'boolean')    ➔ (tt, 'boolean')
    true                                     ➔ 'never-false'
```

This definition may be said to be consistent with Kleene's definition of conjunction in the sense that

ff and ee = ee and ff = ff

The existential quantification is defined in an analogous way:

yo-exists-in-li : Yoke ⟼ Yoke

```
yo-exists-in-li.yok.tyd =
    tyd : Error                                ➔ tyd
    sort-td.tyd ≠ 'L'                          ➔ 'list expected'
    let
        (lis, ('L', typ)) = tyd
    lis = ()                                   ➔ (ff, 'boolean')
    let
        (dat-1,…,dat-n) = lis
        tyd-i          = yok.(dat-i, typ)                              for i = 1;n
    (∃ 1 ≤ i ≤ n) tyd-i = (tt, 'boolean')    ➔ (tt, 'boolean')
    (∀ 1 ≤ i ≤ n) tyd-i = (ff, 'boolean')      ➔ (ff, 'boolean')
    true                                       ➔ 'never-true'
```

Also this definition may be seen as consistent with the Kleene's alternative where

tt kl-or ee = ee kl-or tt = tt

Quantifiers for arrays are defined in an analogous way. Why we assume Kleene's calculus for yokes, rather than the calculus of McCarthy[34], may be justified by an example of an array a = [1/0, 2/1] and a yoke (in an anticipated syntax):

exists-in-ar.(1/(a.i) > 0**)**

which expresses the fact that there exists an element a.i of a such that 1/a.i > 0. In McCarthy's calculus, the value of this yoke would be an error since the (equivalent) alternative

1/a.1 > 0 **mc-or** 1/a.2 > 0

evaluates to error, whereas in the calculus of Kleene it evaluates to tt. Besides, in the calculus of Kleene alternative and conjunction are commutative (except for errors), whereas in the McCarthy's case they are not.

In the end, one methodological remark is in order. The similarity of yoke expressions to value expressions (Sec. 6.4.1) might suggest that yokes could be simply defined as the latter. In this case, however, states would carry expression denotations, and these denotations would take states as arguments, leading to an illegal domain recursion.

4.5 Values, references, objects, deposits and types

Two major concepts that we discuss in this section are *values* and *references*. As already announced in Sec. 4.3, there are two categories of values: typed data and *objects*. Values may be:

- returned by value expressions (Sec. 6.4.2),
- assigned to references in deposits (in states, cf. Sec. 5.3),
- passed to procedures as the values of actual value-parameters (Sec. 6.6.3.4).

---

[34] This calculus will be used in the algebra of expression denotations in Sec. 6.4.2 since at that level Kleene's calculus is hardly implementable.

References are pairs consisting of a *token*, representing some memory location, and a *profile*. The profile describes the *usability* and the *visibility* of the reference (Sec. 5.4). The former determines properties of values which can be stored under this reference, the latter — the rules of accessing them.

The domains of values and references, as well as their related domains, are defined by the following equations:

val : Value = TypDat | Object values
obj : Object = Objecton x ObjTyp objects
obn : Objecton = Identifier ⇒ Reference objectons
typ : ObjTyp = Identifier object types
ref : Reference = Token x Profile references
tok : Token = … (e.g. memory locations) tokens
prf : Profile = Type x Yoke x OriTag profiles
typ : Type = DatTyp | ObjTyp types
yok : Yoke = TypDat ⟼ BooValE yokes
ota : OriTag = Identifier | {$} origin tags
dep : Deposit = Reference ⇒ Value deposits

An *object* is a pair (obn, typ) that consists of an *objecton* and an *object type*. The latter is an identifier which is supposed to be a name of a class (Sec. 5.2). Objects may be said, therefore, to be typed objectons.

An objecton may be regarded as a memory structure whose fields, i.e., references, are bound to identifiers that we shall call *attributes*.

A *reference* ref =(tok, prf) is a pair consisting of:

- a *token* tok, that represents a memory location,
- a *profile* prf = (typ, yok, ota), that determines the way in which ref may be used:
  - type typ determines the *usability* of ref by indicating the type of values that may be assigned to this reference in deposits,
  - yoke yok determines the *usability* of ref by indicating other properties of values assignable to this reference,
  - *origin tag* ota determines the *visibility status* of ref — if it is $ then the reference is public and otherwise it is private (details in Sec. 5.4.3); we assume that $ does not belong to Identifier, and we call it a *public-visibility tag*.

Deposits describe memory contents, since they assign values to references. We will make sure that references in the domain of each deposit that can be built carry distinct tokens.

In the sequel (Sec. 5.3) each memory state will carry an objecton and a deposit. If a reference assigned to an identifier in an objecton does not belong to the domain of deposit, then the identifier is said to be *declared* but *not initialized*, and its reference is said to be a *dangling reference*.

We introduce a special notation and terminology to be used in talking about objects. Consider an objecton obn, a deposit dep, and an identifier ide. We write then:

ide → ref and we say that ide *points to* ref, if obn.ide = ref,

ref → val and we say that ref *points to* val, if dep.ref = val,

Note that we are not talking here about a reference pointing to a location (which is a typical use of references or pointers in programming) but about an identifier pointing to a reference which in turn denotes the memory location at which the identifier's value is being stored. Three situations are possible:

ide → ref → val	a standard situation where ide has been declared and initialized; in this case we say that val is the *value* of ide,
ide → ref	ide has been declared but not initialized; i.e., ref is a dangling reference,
ref → val	no identifier points to ref; in this case we say that ref is an *orphan reference*; such references may appear for instance when we create a local initial store of a procedure call (Sec. 6.6.3.4), and when we return from a local terminal store of a procedure call to a global terminal store (Sec. 6.6.3.5).

References that belong to the range of an objecton, are said to be *carried* by this objecton, and by objects that include this objecton.

The profile, the type, and the origin tag of a reference are also said to be, respectively, the profile, the type, and the origin tag of an attribute that points to this reference.

We extend to values the function that returns the sort of a typed data:

sort-va.val =
  val : TypDat ➔ sort-td.val
  val : Object ➔ 'object'

In a certain sense objectons may be seen as "multireferences" because each of their attributes points to a reference. Note also that these references may point to other objects that carry further references, etc. Consequently, objects may represent nested structures. We see such a situation in Fig. 4.5-1, where no1, ob1,… are attributes, nr1, or1,… are references, A is a metaname of an objecton, B and C are metanames of objects, ClassB and ClassC are names of classes (identifiers), i.e., are the types of corresponding objects.

To the category of types we add a constructor of object types, which from a set-theoretical perspective is an identity function, but from an algebraic perspective it is not, because it "makes identifiers to be object types".

ty-create-ot : Identifier ⟼ ObjTyp
ty-create-ot.ide = ide

Such an algebraic constructor is called an *insertion.*

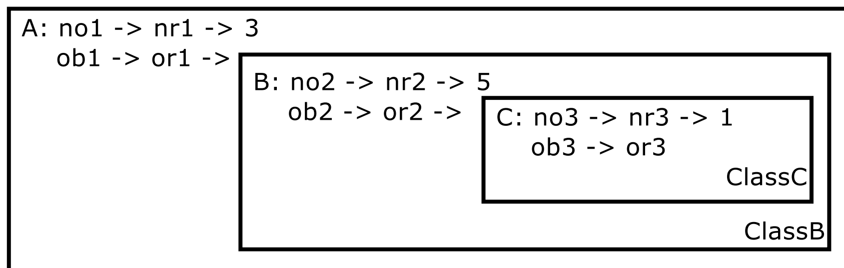


Fig. 4.5-1 A structure view of an objecton

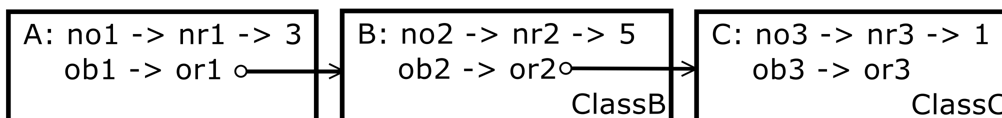


Fig. 4.5-2 A graph view of an objecton

It is to be noted that structured types, i.e. array-, list- and record types always belong to the category of data types. We do not introduce constructors to build structured types involving object types, such as, e.g., types of lists or arrays of objects, which is a consequence of the decision not to deal with structured values such as, e.g., lists or arrays of objects. We refrain from discussing these options just for the sake of simplicity and brevity. As we are going to see in Sec. 6.6.5.3, new objects will be built exclusively by *object constructors*, and by instructions that modify earlier constructed object.

Nevertheless, we can build objects that may be colloquially called “lists of objects”, but their “list nature” is just a way of seeing them.

Consider an objecton in Fig. 4.5-1. Its visualization will be referred to as *structure view* of this objecton. An alternative to it is a *graph view* — in this example a *list* view — shown in Fig. 4.5-2., but it is not a list of objects. In our model we only have values that are lists, but we do not have lists of values.

If an object B is assigned to an attribute of an object A, as in our example, then we say that B is an *inner object* of *depth 1* of A. The inner objects of B will be also regarded as inner objects of A, but of a deeper depths and so on. The attributes of A will be called *surface attributes* of A, whereas all attributes of B and C will be called *deep attributes* of A.

In the sequel we shall carefully distinguish between an *object attribute*, that is analogous to *integer attribute*, and that is an attribute whose value is an object, and *an attribute of an object*, or *object's attribute*, that is an attribute in the objecton of an object.

As we are going to see, it may be convenient to regard a value as a pair consisting of an element that we shall call *the core of the value,* and a type:

```
val : Value    = Core x Type
cor : Core     = Data | Objecton
```

5 CLASSES AND STATES

5.1 Classes intuitively

Classes may be regarded as collections of tools used to create and modify objects. They carry three categories of tools:

1. types (maybe none),
2. methods (maybe none), which are either signatures of procedures, or so called pre-procedures, and which include two subcategories:
   a. imperative pre-procedures, functional pre-procedures and their corresponding signatures,
   b. object constructors and their signatures,
3. one objecton (maybe empty) used as a pattern for all objects generated from this class.

Let's forget for a moment about types and methods, and concentrate on objectons carried by classes. Consider the following class declaration written in an anticipated syntax[35] of **Lingua**.

```
class CartesianPoint
   let abscissa = 2,15 be real and public tel;
   let ordinate be real and private tel
ssalc
```

The fact that abscissa is initialized to 2,15 only means that if we generate an object directly from the class, then abscissa will be initialized to 2,15 but later we can change its value. In turn the attribute ordinate may be (but do not need to) left not initialized. Consequently an object of type CartesianPoint may be of the form:

'abscissa' → (ab-tok, ('real', TT), '$') → (2,15, ('real', TT))
'ordinate' → (or-tok, ('real', TT), 'CartesianPoint')
('CartesianPoint')

or of the form

'abscissa' → (ab-tok, ('real', TT), '$') → (3,16, ('real', TT))
'ordinate' → (or-tok, ('real', TT), 'CartesianPoint') → (4,75, ('real', TT))
('CartesianPoint')

In the first case the reference of 'ordinate' is dangling which expresses the fact that this attribute has been declared but not initialized. In the second case it has been initialized to a real value. Since this attribute has been declared as private its origin tag is 'CartesianPoint'.

In our example both objects of class CartesianPoint are of the same shape. The situation complicates, when we introduce recursion to the definitions of a class. Let's consider the following class declaration written again in an anticipated syntax. In this case it includes three declarations:

1. of an objecton's pattern,
2. of a special-purpose functional procedure called an *object constructor,*
3. of an imperative procedure which calls the object constructor and modifies global memory states.

Below we see an example of a program with one class declaration, and one call of a procedure that builds a circular object. This procedure calls an object constructor that belongs to a special category of procedures.

---

[35] In a general case declarations of variables indicate types, yokes and privacy status of these variables. At the level of colloquial syntax we assume that if the yoke is TT (always true) then we may skip it in the declaration.

```
class ListNode

    let no = 23 be integer and public tel;
    let next  be ListNode and public tel

    cons ConstructObject(val number as integer, node as ListNode return ListNode)
        no      := number + 1;
        next    := node
    snoc

    proc BuildCircularList()
        let i be integer  tel;
        let node be ListNode tel;
        i := 1;
        while i <= 3
            do
                node  := ListNode.ConstructObject(i, node);
                i     := i+1
            od;
        node.next.no     := 11
        node.next.next  := node;
    corp

ssalc;
ListNode.BuildCircularList()
```

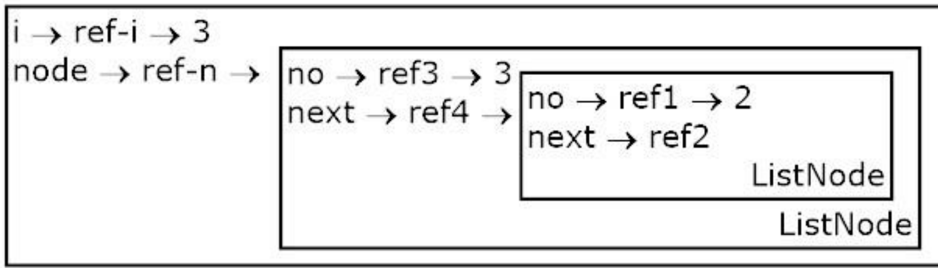
In this example the attribute next is of the type of the class which is just being declared. The execution of BuildCircularList() generates the following sequence of objectons, where the first objecton results from the execution of the declaration of local attributes of this procedure:

```
i → ref-i → 1
node → ref-n
```

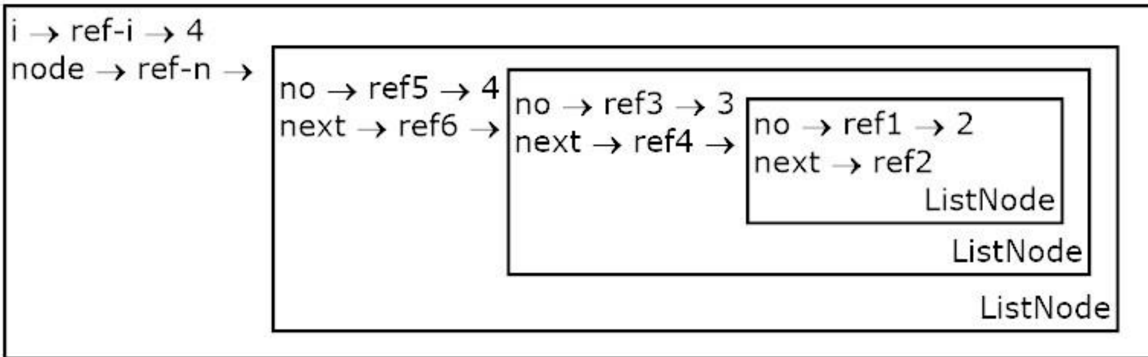
In the next step our procedure enters the **while** loop, and there calls the object constructor ConstructObject and passes to it two actual value parameters: i of value 1, and node with a dangling reference. The object constructor builds an object by copying (with new references) and modifying the objecton of the class. Then the new object is assigned to node and the value of i is augmented by one.

```
i → ref-i → 2
node → ref-n → [ no → ref1 → 2
                 next → ref2
                              ListNode ]
```

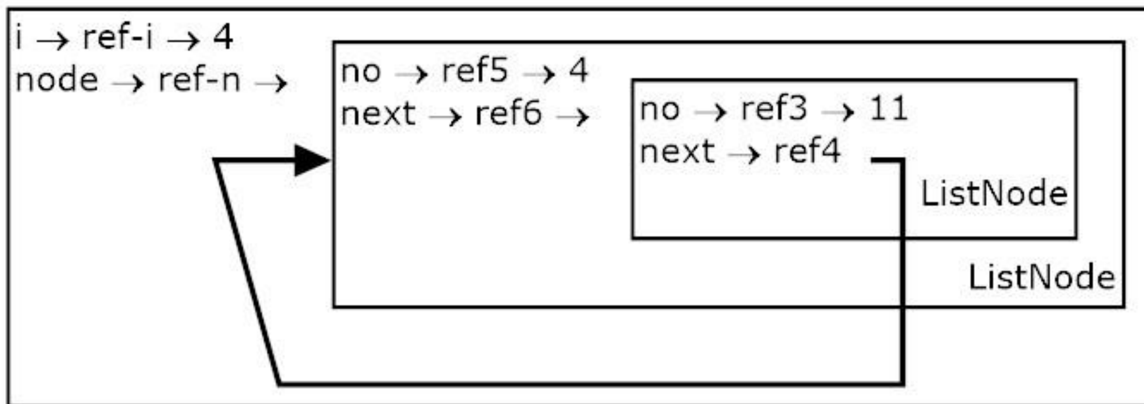
In the following step we assign to node a modified class objecton where the formerly created object is assigned to next:



This action is repeated thus producing the following objecton. Our program exits the loop.



In the last step our program performs two last assignments which in the deepest object modifies the value of no, and redirects the reference of the deepest next to the surface objecton. In this way we have constructed a circular object. The rules of building objects from classes are formalized in Sec.6.6.5.



At the end of this section let's list assumptions about classes and objects in our model that we have adopted to make it possibly free from technical complications:

1. We do not introduce neither packages nor compilation units.
2. Classes do not include inner classes.
3. As a consequence of 1. and 2. all classes are public.
4. Classes do not contain not-replicable attributes.
5. Types and methods are declared exclusively in classes and are public.
6. A class once declared is never changed, but can be copied and then modified to build a new class (heritage).
7. Objects are created exclusively by object constructors that replicate class objectons.
8. Some attributes of object may be private; for such attributes, if we wish to provide an external access to them we have to declare appropriate **getters** and/or **setters** in the corresponding class.

5.2 Classes formally

By a *class* we shall mean a tuple consisting of four elements: an identifier, two mappings (possibly empty) — a *type environment*, and a *method environment* — and one objecton (possibly empty):

cla : Class = Identifier x TypEnv x MetEnv x Objecton classes
tye : TypEnv = Identifier ⇒ Type | {Θ} type environments
mee : MetEnv = Identifier ⇒ Method method environments
met : Methods = ProSig | PrePro methods

where Θ is a special element called a *pseudotype.* The domains ProSig of *procedure signatures*, and PrePro of *pre-procedures,* will be defined in Sec. 6.6. Each class is, therefore, a tuple of four elements:

(ide, tye, mee, obn).

By an *empty class* we mean a class where all three mappings are empty:

(ide, [ ], [ ], [ ]).

Identifiers bound to types will be called *constants,* since their values, once assigned to them, are never changed. Identifiers bound to values (though references) in state objectons, but not in class objectons, will be called *variables*, since their values may be modified. The first element of a class, the identifier, is called an *internal name of a class*. We will see why we need these internal names in Sec. 6.7.4.2, where we describe an action of adding a new attribute to (the objecton of) a class.

5.3 Stores and states

The domain of states is defined as follows:

sta : State = Env x Store states
env : Env = ClaEnv x ProEnv x CovRel environment
cle : ClaEnv = Identifier ⇒ Class class environments
pre : ProEnv = Indicator ⇒ Procedure procedure environment
ind : Indicator = Identifier x Identifier indicators
sto : Store = Objecton x Deposit x OriTag x SetFreTok x (Error | {'OK'}) stores
cov : CovRel = Sub.((DatTyp x DatTyp) | (ObjTyp x ObjTyp)) covering relations
sft : SetFreTok = Set.Token sets of (free) tokens

The environment of a state carriers classes, procedures[36] (Sec. 6.7.6), and covering relations[37] (Sec. 5.4.2), and the store carries the rest. Classes and types declared in them are going to be public, whereas their methods and attributes are going to be private[38]. Also the attributes carried by objectons of stores will be private. The meanings of "public" and "private" are explained in Sec. 5.4.3.

If a class is assigned to an identifier in a class environment, then we say that this identifier *points to* this class. In an analogous way we talk about identifiers pointing to types in the type environments of classes, and to pre-procedures in procedure environments.

Identifiers that point to classes in states are called the *external names* of these classes, and the corresponding classes are said to be *declared* in sta.

---

[36] We recall that classes carry pre-procedures rather than procedures. The difference between these two concepts is explained in Sec. 6.6.

[37] The decision of putting covering relations in environments is technically not especially relevant. We just decided to "keep them in the same place", where we keep classes and their types.

[38] This decision has an editorial character and serves the technical simplification of our model.

Attributes that appear in objectons of classes are called *class's attributes*. A surface attribute of the objecton of a store is traditionally called a *variable* in this store and state respectively.

An object of the form (obn, MyClass), where obn has been created by an object constructor (see Sec. 6.6.5.2) from the objecton of cle.MyClass, is said to be an *object of class* MyClass.

The covering relations cov between types will be used to describe a *usability regime* defined in Sec. 5.4.2.

The origin tag that appears in a store is called the *origin tag of the store,* and of the hosting state as well. Its role will be explained in Sec. 5.4.3, where we shall talk about a *visibility regime*.

The sets of free tokens will be used to provide "fresh" (not used) tokens, for the declarations of value variables and the constructors of new objects. For that sake we assume the existence in our model of a function:

get-tok : SetFreTok ⟼ Token x SetFreTok
get-tok.sft = (tok, sft − {tok})) such that tok : sft

An objecton my-obn is said to be *well-formed* in a state sta = ((cle, pre, cov), (obn, dep, ota, sft, err)), if:

1. for any attribute ide, if obn.ide = !, and dep.(obn.ide) = !, then:
   obn.ide **VRA.cov** dep.(obn.ide) — value by reference acceptability (see Sec. 5.4.2),
2. all inner objectons of obn are well-formed in sta.

A class (ide, tye, mee, obn) is said to be *well-formed* in a state, if

1. obn is well-formed in this state,
2. for every reference (tok, (typ, yok, ota)) in obn, its origin tag ota is either $ or ide[39].

A state sta = ((cle, pre, cov), (obn, dep, ota, sft, err)) said to be *well-formed*, if:

1. obn is well formed in sta,
2. external names of all classes declared in cle coincide with their internal names,
3. all surface and inner objects in obn are of types that are the names of classes declared in cle,
4. all classes declared in cle are well-formed,
5. sft includes only such tokens that do not appear in references bound in dep,
6. every identifier appearing in a state, appears in it only once; e.g., if an identifier is a variable, it can't by at the same time a type constant or a class name.

As we see, the well-formedness of states is mainly about typing. In the sequel, we shall ensure that the states appearing in the executions of our programs are well-formed. By:

WfState

we denote the sets of all well-formed states. For technical convenience we define the following auxiliary functions:

error : Store ⟼ Error
error.(obn, dep, ota, sft, err) = err

error : State ⟼ Error
error.(env, sto) = error.sto

Formally we may assume that the function error is defined on the union Store | State. In the same spirit we define next two functions:

is-error : Store ⟼ Boolean
is-error.sto =
  error.sto ≠ 'OK' ➔ tt
  true ➔ ff

is-error : State ⟼ Boolean
is-error.(env, sto) = is-error.sto

Again, not quite formally, we define a function on the domain (State x SetFreTok) | (State x Error):

◄ : State x Error ⟼ State

---

[39] This definition is the main cause why we have introduced internal names of classes as the elements of classes (see also Sec. 6.7.4.2). An alternative solution might be to talk about the well-formedness of classes only in the context of states, that, in our opinion, would be less elegant.

(env, (obn, dep, ota, sft, err)) ◄ new-err = (env, (obn, dep, sft, ota, new-err))

◄ : State x SetFreTok ⟼ State
(env, (obn, dep, ota, sft, err)) ◄ new-sft = (env, (obn, dep, new-sft, ota, err))

We also assume that this function will be applicable to stores in an obvious way. We shall use a function:

declared : Identifier x State ⟼ {tt, ff}

to protect us against double declaration of some identifiers. In an obvious way we extend this function to stores.

By an *empty state* we shall mean every state of the form (([ ], [ ], Ld-cov), ([ ], [ ], 'public', { }, 'OK')) where Ld-cov is a covering relation defined by language designer (see Sec. 5.4.2). As is easy to check, empty state is well formed.

5.4 Two regimes of handling items

5.4.1 An overview

By an *item,* we shall mean a value, a reference, a type, a method (Sec. 6.6), or a class. All items are storable, and we access them through *indicators* that are tuples of identifiers. To indicate a class, we need one identifier, to indicate a type or a procedure, we need two identifiers — one for a class plus one for the type/procedure itself — to indicate a value or a reference of a variable, we need one identifier, but in the case of object attributes, we may need more identifiers, if such an attribute is located at a deep level of an object.

The principles of accessing and using items will be described by two systems of rules that we shall call *handling regimes*:

- A *usability regime* is described by means of the types of values and the profiles of references, and serves the purpose of deciding which values can be "sent" to a chosen operator as its arguments, or can be assigned to a chosen reference in a deposit. E.g. we can't "send" real numbers to an integer division, or assign a negative integer to a variable whose type is 'integer', but whose yoke requests that the assigned value is positive. Technically usability regime is built into the denotations of expressions, assignment instruction, variable- and attribute declarations, procedure declarations, and procedure calls.
- A *visibility regime* is described by means of the origin tags of references and of states, and serves the purpose of deciding which item indicators are accessible at a given stage of program execution; e.g., we shall assume that private attributes of a class will be visible exclusively in the bodies of procedures declared in this class. Technically, a reference, to be visible in a state must have the origin tag identical with the tag of the state (cf. assignment instructions in Sec. 6.4).

Note that pre-procedures are not regarded as items. They will not be accessible from syntactic level, and will constitute sort of "raw components" used in building procedures. This technique, which is explored in Sec. 6.7.6, has been adopted to describe the execution of mutually recursive procedures declared in different classes. Procedure declarations included in the declarations of classes will first assign pre-procedures to procedure names in method environments of classes, and later a special mechanism of *procedure opening* (Sec. 6.7.6) will assign procedures to their indicators in procedure environments of states. Although formally procedure declarations in classes build pre-procedures, we shall talk, for simplicity, about procedures declared in classes.

Note a significant differences between two described regimes — usability is a property of values, whereas visibility is a property of the indicators of items. We may colloquially say that a locally declared procedure is visible only in a local state, but precisely speaking, what may be seen or not is the indicator of a procedure rather than a procedure itself.

In two following sections we give a birds-eye view to the ideas of our handling regimes, to be later incorporated in our model.

5.4.2 Usability regime

Basic rules of usability regime are the following

1. Every value includes a type, and every reference includes a profile consisting of a type, a yoke and an origin tag.
2. If a value is going to be assigned to an attribute (via its reference), by a declaration (Sec. 6.7.2), or by an assignment instruction (Sec.6.4), or by a parameter passing mechanism of a procedure (Sec.6.6.3.4), then the type of the attribute must *accept* the type of the value, and the value must satisfy the yoke of the reference. The concept of type acceptance is explained below.
3. If a function (operation) is applied to its arguments, then the types of arguments "expected" by the function must *accept* the types of the current arguments.

To formalize the concept of type acceptance we return to the notion of a *covering relations* with the following domain (Sec. 5.3):

cov : CovRel = Sub.((DatTyp x DatTyp) | (ObjTyp x ObjTyp))

The fact that (typ-1, typ-2) : cov will be also written as typ-1 **cov** typ-2, and will be said that typ-1 *covers*, or *accepts* typ-2. As we see, a data type may cover only another data type, but not an object type, and vice versa. The typesetting of **cov** in bold is just a "meta-syntactic sugar" to make some meta-formulas easier to read.

We assume that each covering relation cov will be partly defined by a language designer, and partly by a programmer. Consequently it will be a union of two (disjoint) relations:

cov = Ld-cov | Pr-cov

where

1. Ld-cov is a component defined by a language designer, i.e. available in all programs of a given language,
2. Pr-cov is a component defined by a programmer, i.e. available exclusively in the program where it has been established.

For instance, a language designer may decide that a data type 'integer' covers data type ('small-integer'), and a programmer — that an object type 'employee-type' covers object type 'accountant'.

Now, for every covering relation cov we define two induced *acceptability relations*:

TTA.cov ⊆ Type x Type   type-by-type acceptability relation

VRA.cov ⊆ Reference x Value   value-by-reference acceptability relation

The first of them is a completion of **cov** to a reflexive and transitive relation, which means that **TTA.cov** is the least relation on types such that

(1) cov ⊆ TTA.cov,
(2) (typ, typ) ⊆ TTA.cov for every typ : Type,
(3) if (typ1, typ2), (typ2, typ3) : TTA.cov then (typ1, typ3) : TTA.cov.

The second induced relation concerns not only a relationship between types but also the satisfaction of the yoke by the value

(tok, (typ-r, yok, ota)) **VRA.cov** (dat, typ-v)
iff
(1) typ-r **TTA.cov** typ-v   and
(2) yok.(dat, typ-v) = (tt, 'boolean')

In Sec. 5.3 we have assumed that the relationship ref **VRA.cov** val must be satisfied in all well-formed states.

In defining the denotational level of **Lingua** we shall give our programmers a tool for the creation of covering relations by enriching current relations by new pairs. For this sake we assume the existence in our model of a function of enrichment with the following signature:

enrich-cov : CovRel x Type x Type ⟼ CovRel | Error

and the following definitional scheme:

enrich-cov.(cov, typ-1, typ-2) =
typ-1 : ObjTyp **and** typ-2 /: ObjTyp ➔ 'typ-2 must be an object type'
typ-2 : ObjTyp **and** typ-1 /: ObjTyp ➔ 'typ-1 must be an object type'
typ-1 = typ-2 ➔ 'equal types can't be used'
typ-1 cov typ-2 ➔ 'redundant definition'[40]
other cases ➔ other error signals
true ➔ cov | {(typ-1, typ-2)}

We leave the "other cases" not specified to avoid going into too many technical details. An example of a pair of types that should be rejected by enrich-cov may be (typ ,('A', typ)).

Note that our constructor does not check if object types added to cov are carrying the names of declared classes. Such a check must refer to a state, and therefore will be introduced at the level of denotations in Sec. 6.7.5.

5.4.3 Visibility regimes

Zadaję sobie pytanie, czy ten rozdział nie powinien być przeniesiony na koniec Sec.6 jako podsumowanie mechanizmów widoczności, gdy czytelnik będzie już wiedział jak działają wywołania procedur? Z drugiej strony jakaś zapowiedź reżymów widoczności jest w tym miejscu chyba potrzebna. ???

From a programmer's perspective, visibility rules explain in which programming contexts a given item indicator may be used (is visible). E.g. we will set a rule that private attributes of a class may be referred to exclusively in the bodies of procedures declared in this class. At the same time, items locally declared in the body of a procedure will be visible exclusively in this body.

In our model of object-oriented languages we will have two orthogonal "dimensions" of *visibility statuses*:

- ***procedure-dependent visibility***: all items locally declared in procedure bodies will be visible exclusively in these bodies,
- ***class-dependent visibility***: selected items in classes may be declared as *private*.

Let's start from the former, and let's anticipate an assumption later formalized in Sec. 6.3, that every program in our model consists of a (possibly composed) declaration followed by a (possibly composed) instruction. This assumption does not limit the expressiveness of programs, but considerably simplifies our model. Under this assumption all global instructions of a program, i.e. all instructions except local instructions of procedure calls, operate on a common *global environment,* and a common *global-state objecton*. It is so, since instructions may only change values assigned to references in deposits.

Assume now that we call an imperative procedure in some current state which we call *initial global state* ig-sta = (g-env, ig-sto), and which consists of a *global environment* and an *initial global store*. In our model the execution of a call consists of three steps:

- First, we create an *initial local state* il-sta = (g-env, il-sto), that consists of a global environment, and an *initial local store*. The latter binds (in the objecton) only formal parameters. At the same time it inherits the whole deposit from the global store. Formal reference-parameters point directly to the refer-

[40] Mathematically we could have assumed that in this case the enrichment operation returns an unchanged cov relation. However, if a programmer tries to add a pair of types that is already in cov, then they probably do it by mistake, and therefore such fact should be signalized by the system.

ences of actual-reference parameters, and the references of the remaining global variables become orphans (Fig. 6.6-2 in Sec. 6.6.3.4)

- Next, we execute the body of the procedure thus transforming the initial local state into a *terminal local state* tl-sta = (tl-env, tl-sto). Since the bodies of procedures may be arbitrary programs, in the course of their executions local classes, procedures and variables may be declared.
- At the end of the call, we exit from the call, and create a *terminal global state* tg-sta = (g-env, tg-sto), where we regain the global environment, and global deposit (Sec. 6.6.3.6). At this stage all locally declared classes, procedures, and variables cease to exist, and therefore are no more visible.

Note now, that in the above anticipation of the mechanism of imperative procedures we have made three important decisions concerning global visibility:

- all global classes are visible in all local states of procedure calls, since they are declared in the global environment,
- the references of all actual reference parameters are visible in local stores,
- all locally declared classes and variables cease to exist after the termination of the call.

All these decisions have an engineering character, since from a mathematical perspective, we could have decided differently, e.g. that locally declared items remain visible after exiting a procedure call, or that only some of the globally declared items are visible locally.

As we are going to see in Sec. 6.7.4.6, our mechanism of public visibility is even more complicated, since in procedure declarations we accept an *anticipated visibility* of procedures that haven't been declared yet. At the same time, however, in the case of types we require an *ex post visibility*, which means that a type must be declared to be visible.

Let us proceed now to visibility statuses. We assume — again for the simplicity of our model — that private may be only the attributes of classes and objects, if they are declared to be so, whereas all other items are always public.

Variables, classes, types and procedures are always public.

The visibility status of attributes is established in a class declaration, and later is inherited by all objects of this class.

Technically the visibility status of an attribute is in fact the visibility status of its reference, and the latter depends on the origin tag of the reference according to the following general rules:

General visibility rules

1. A reference is visible in a state, if the origin tag of this reference
   1.1. either is $, or
   1.2. coincides with the origin tag of the state.
2. A reference must be visible whenever we intend to:
   2.1. get a value assigned to it in evaluating an expression,
   2.2. change the value assigned to it in executing an assignment instruction.
3. The origin tags of references and states are established when these references and states are created, and later they can't be changed.

We will say that a variable is *declared* in a state, if it is bound in the objecton of this state.

If a variable has been declared in a state, then we shall say that this state is a *hosting state* of this variable and ot its value. If we declare an attribute in a class, then we say that this class is a *hosting class* of this attribute and of its value.

If an origin tag of an attribute is a name of a class, then this attribute is said to be *private for this class*. Such an attribute will be visible only in states whose origin tag is the name of the class. As we are going to see, the only such states will be the local state of the calls of procedures declared in this class.

Operational visibility rules

Let MyClass be a class named 'MyClass', and let myProc be an arbitrary procedure declared in this class, and named 'myProc'.

4. When we declare an attribute in a class called 'MyClass', then:
   4.1. if this attribute is to be private, then its origin tag is set to 'MyClass',
   4.2. if this attribute is to be public, then its origin tag is set to $.
5. When we declare a variable in a state, then it is always public, and therefore the origin tag of its reference is $. Note that a variable which is local to a procedure call is "locally public" for this call.
6. Local states of a call of myProc have origin tags 'MyClass', which means that private attributes of type 'MyClass' are visible in these states. Of course, public attributes are visible by principle.
7. When we pass an actual value parameter to a procedure call, then the reference of the corresponding formal parameter gets the yoke and the origin tag of the reference of the actual parameter (visibility inherited),
8. When we pass an actual reference parameter to a procedure call, then its reference becomes the reference of the corresponding formal parameter (visibility is inherited).

Note that rule 6. allows for the construction of dedicated procedures, traditionally called *getters* and *setters*, to be used when we wish to reach private attributes of objects. This rule is "implemented" in the constructors of the denotations of value expressions (Sec. 6.4.2), and reference expressions (Sec. 6.4.4).

To go deeper into the details of these rules let's analyze an example illustrated in Fig. 5.4-1. Consider a program that we shall call the *main program*, and assume that the execution of this program starts in a state whose origin tag is $. Since, as we are going to see in Sec. 6.5, instructions may only change values assigned to attributes (but not their references), starting from the instruction of the main program, all states will have a common fixed environment, let's call it *global environment*, and a common fixed objecton, let call it *global objecton*. Together they will be components of all consecutive *global states* and their *global stores*. In our example the global state is the following:

- In the global environment it binds three items:
  - A class MyClass named 'MyClass' with:
    - a private class attribute 'att1' with origin tag 'MyClass',
    - a public class attribute 'att2' with origin tag $,
    - a pre-procedure myProc named 'myProc',
    - an object pre-constructor named 'myCons',
  - A procedure myProc named 'myProc'; its indicator is a pair of identifiers ('MyClass', 'myProc'), but for the lack of space on the picture we show only one of them,
  - An object constructor named 'myCons'; object constructors belong to the category of procedures (a comment as above),
- In the objecton it binds
  - two (public) object variables 'myObj1', and 'myObj2' pointing to objects of type 'MyClass'. As a rule, the objectons of these objects have been generated by an object constructor; note that each of them has one public attribute and one private one,
  - a public variable 'var'.

Now, assume that one of the instructions of our main program is a call of the procedure myProc with one value parameter, and reference parameter. This call creates a local state, and applies to it a program, that is the body of the procedure. The initial local state of this program includes (Sec. 6.6.3.2):

- a global environment inherited from the global state (we do not show it in the figure),
- a new local store with the origin tag that is the name of the class, where myProc was declared — in our case it is 'MyClass'; the latter fact makes visible in this state all private attributes with origin tag 'MyClass'.

According to the rules of passing actual parameters to formal parameters, described in Sec. 6.6.3.4, the objecton of the local store, let's call it *local objecton*, binds two formal parameters. We assume additionally that the program of the body includes a declaration of a local variable 'loc-var'. Altogether the local objecton binds, therefore, three public variables:

- a *local variable* 'loc-var' with origin tag $, declared in the procedure body according to the rule 5.,
- a *formal value-parameter* 'for-val-par' pointing to a reference with a fresh token and the profile of *actual value-parameter* 'myObj1', that points to a twin of the value of 'myObj1',
- a *formal reference-parameter* 'for-ref-par' pointing to the reference of *actual reference-parameter* 'myObj2',

Now, let's analyze the *visibility perspectives* of our main program that operates on the global state, and of the program that constitutes the body of our procedure, and operates on the local state:

The visibility perspective of the main program:

- The globally declared class and both procedures.
- Two globally declared (public) object-variables 'myObj1' and 'myObj2'. We can assign their values to another (public) variable, we can assign a new value to this variable, and we can pass this variable as an actual parameter to a procedure call. However, we can't reach the private attribute 'att1' of either of these objects, unless by a dedicated procedure (getter or setter) declared in MyClass. In our case this is myProc.
- One globally declared (public) variable 'var'.

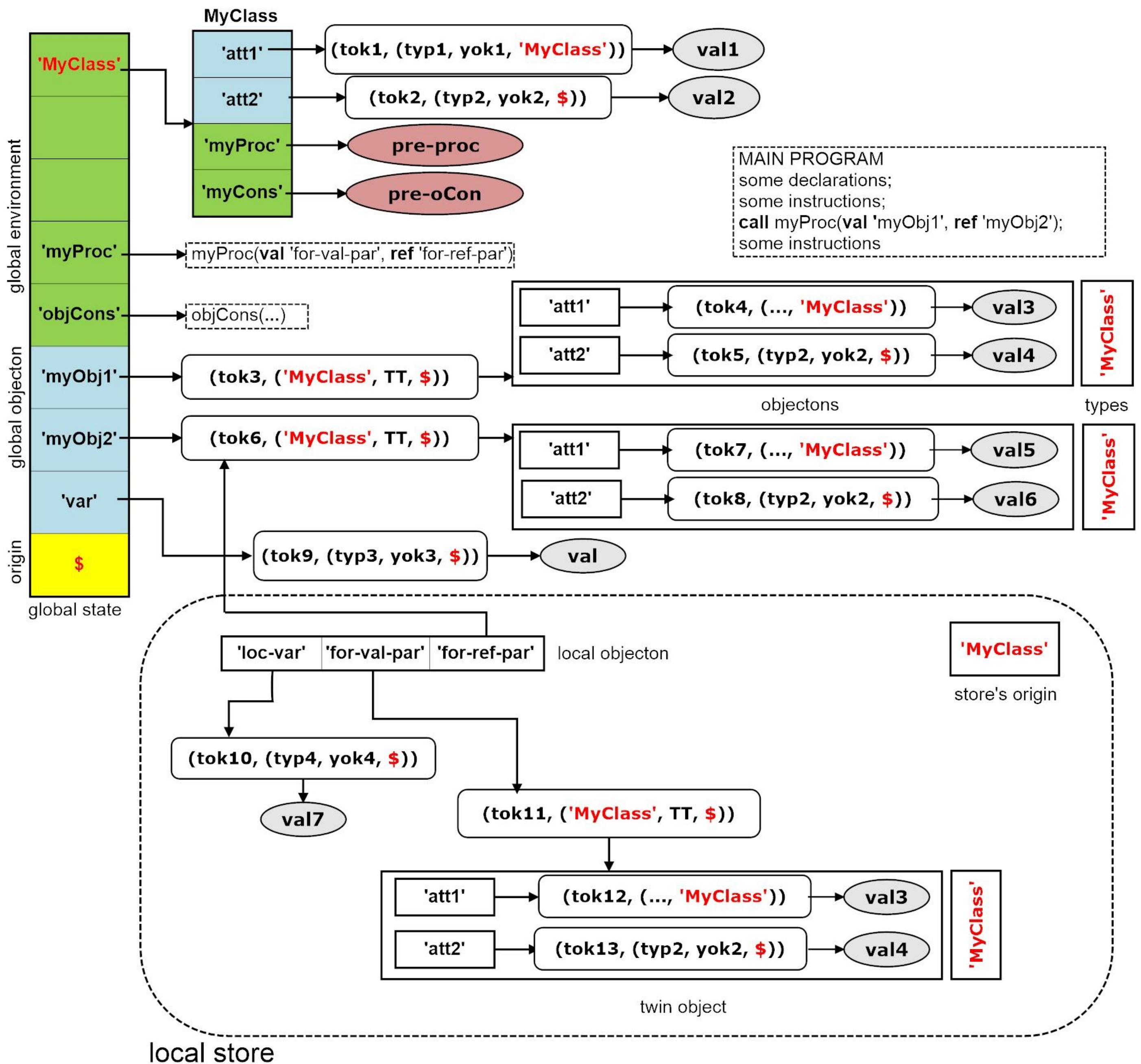


Fig. 5.4-1 An illustration of the visibility concept

The visibility perspective of the body program:

- All globally and locally declared classes and procedures.
- One local variable 'loc-var' declared in the body program.
- One local object-variable 'for-val-par' that points to a twin of the value of actual parameter 'myObj1'.
- One local object-variable 'for-ref-par' that points directly to the reference of 'myObj2' in the (inherited by the local store) global deposit.

Let's note at the end that when we call locProc, then the local state of myProc becomes for locProc a global state with all the consequences of this fact, but with one exception — its origin tag is 'myClass' rather than $. At the same time, however, all variables bound at this level, either locally declared, or passed as parameters, are public, and their visibility status is inherited by the local states of all levels of the locality.

6 DENOTATIONS

6.1 The carriers of the algebra of denotations

The denotations of our language will constitute an algebra of denotations **AlgDen** that will become a component of the diagram of algebras described in Sec. 3.4. The carriers of this algebra are the following:

Primitive carriers

ide	: Identifier	= …	identifiers
prs	: PriSta	= {'private', 'public'}	privacy statuses indicators
loi	: ListOfIde	= Identifier^{c*}	lists of identifiers
cli	: ClaInd	= {'empty-class'} | Identifier	class indicators

Applicative carriers[41]

yok	: YokExpDen	= WfState → YokeE	yoke-expression denotations
ted	: TypExpDen	= WfState ↦ TypeE	type-expression denotations
ved	: ValExpDen	= WfState → ValueE	value-expression denotations
red	: RefExpDen	= WfState ↦ ReferenceE	reference-expression denotations

Imperative carriers

dcd	: DecDen	= WfState → WfState	declaration denotations
pod	: ProOpeDen	= WfState ↦ WfState	procedure opening denotation
ctc	: ClaTraDen	= Identifier ↦ WfState → WfState	class-transformation denotations
ind	: InsDen	= WfState → WfState	instruction denotations
ppd	: ProPreDen	= WfState → WfState	program-preamble denotations
prd	: ProDen	= WfState → WfState	program denotations

Declaration-oriented carriers

dse	: DecSec	= ListOfIde x TypExpDen	declaration sections
fpd	: ForParDen	= DecSec^{c*}	formal-parameter-denotations
apd	: ActParDen	= ListOfIde	actual-parameter-denotations

Signature carriers

ips	: ImpProSigDen	= ForParDen x ForParDen	imperative-procedure signature denotations
fps	: FunProSigDen	= ForParDen x TypExpDen	functional-procedure signature denotations
ocs	: ObjConSigDen	= ForParDen x Identifier	object-constructor signature denotations

The denotations of value expressions, declarations, instructions and programs are partial functions since all of them may generate infinite executions.

Type expressions are used in the declarations of types, variables, class attributes and methods. The role of the domain of program preamble denotations will be explained in Sec. 6.3

---

[41] In early programming languages used at the turn of the 1940s and 1950s, programs were sequences of simple instructions called "commands"; therefore, they could be said to be written in an "imperative mood". Complex expressions came into play later in higher-order languages such as Algol and Fortran. Expressions were regarded as tools to be "applied" to get values. Over time, languages (practically) without instructions, i.e., where programs were expressions, started to emerge and were called "applicative languages". One of the first was Lisp — a language for manipulating lists.

Note that the domains of items — i.e., of values, references, types, yokes, procedure and classes — are not the carriers of our algebra, which means that they will not have their counterparts in the algebras of syntax. Values, references, types and classes will be (indirectly) represented by expressions, and procedures — by declarations and calls.

An applicative denotation is said to be *conservative*, if given a state that carries an error, returns this error as a result. A constructor of applicative denotations is said to be *diligent,* if given conservative denotations as arguments return a conservative denotation as a result. As we are going to see, all applicative denotations reachable in our language will be conservative.

An imperative denotation is said to be *conservative,* if given a state that carries an error, returns the same state as a result. A constructor of imperative denotations is said to be *diligent*, if given conservative denotations as arguments returns a conservative denotation as a result. Typical imperative denotations in **Lingua** will be conservative which implementationally means that once an error message is raised during the execution of a program, the execution aborts and the error message is signalized. However, our model of errors allows for an introduction of error-handling mechanisms, where occurrences of errors trigger recovery actions. An example of a corresponding not-diligent constructor of instruction denotations is discussed in Sec. 6.5.3.

In the subsections that follow we shall define the constructors of our algebra of denotations.

6.2 Identifiers, class indicators and privacy statuses

Identifiers, class indicators and privacy statuses have a singular character in our model since they are common for the algebras of denotations and syntax. We decided (a mathematical decision) that talking about the "denotations of identifiers" on one hand, and of "syntax of identifiers" on the other, and similarly for two remaining categories, would be a too-far going fundamentalism. We assume, therefore, that their denotational carriers and their abstract-, concrete- and colloquial syntactic carriers are the same, and are just sets of strings of characters.

Identifiers are algebraically built by zero-argument constructors, one for every identifier. Each of them makes an identifier "out of nothing":

build-id-ide.() = ide for every ide : Identifier

Here () denotes an empty tuple of arguments.

We recall in this place (cf. Sec. 2.13) that the future algebra of syntax of our language will be constructed as a homomorphic co-image of the (unique) reachable subalgebra of **AlgDen**. Consequently, only reachable denotations will have their counterparts at the level of syntax. Class indicators will be generated by two constructors:

create-class-ind-of-empty : ⟼ ClaInd
create-class-ind-of-empty.() = 'empty-class'

create-class-ind-of-parent : Identifier ⟼ ClaInd
create-class-ind-of-parent.ide = ide

Their role will be explained in Sec. 6.7.3. Privacy statuses are also built by two constructors:

build-ps-private : ⟼ PriSta
build-ps-private.() = 'private'

build-ps-public : ⟼ PriSta
build-ps-public.() = 'public'

The role of these elements was explained in Sec. 5.4.

6.3 Programs and their segments

Before we proceed to the denotations of expressions, instructions and declaration we shall take a "strategic" decision about the future syntax of our programs. We presume that they will consist of three *segments* sequentially composed in this order:

1. a *preamble* consisting of (sequentially interleaving) instructions and declarations,
2. one universal procedure-opening command **open procedures**,
3. an instruction.

Of course, all mentioned above instructions may be structured, i.e. including other instructions. Besides, the first and the third segment may be trivial skip-segments.

As we will see, the procedure-opening command **open procedures** is a special tool for the elaboration of recursive procedures whose declarations may belong to different classes (details in Sec. Sec. 6.6.1 and 6.7.6).

The assumed restriction of the structure of programs has partly engineering and partly mathematical justification.

At the engineering side it should help programmers to better understand and control the "behaviors" of their programs. Declarations build tools to be used in programs, and therefore it seems reasonable to start from them in developing a program. In turn, instructions included in preambles are necessary for building values, and in particular objects, to initialize declared variables and attributes .

At the mathematical level our assumption will simplify the mechanism of passing returning the references of formal reference-parameters of procedure calls (Sec. 6.6.3.5) and consequently also the rule of building correct procedure calls (Sec. 9.4.6.3). It also standardizes the process of correct program development (Sec. 9.4.1).

So far, our assumption about programs' structure was described at the level of syntax[42]. To bring it to denotations we introduce the following constructor:

make-prog-den : ProPreDen x {open-pro-den} x InsDen ⟼ ProDen
make-prog-den.(ppd, pod, ind) = ppd ● open-pro-den ● ind

where

open-pro-den : WfState → WfState

is a denotation of command **open procedures** defined in Sec. 6.7.6. We also define three constructors of the declarations of program preambles:

make-ppd-of-dcd : DecDen ⟼ ProPreDen — an insertion
make-ppd-of-dcd.dcd = dcd

make-ppd-of-ind : InsDen ⟼ ProPreDen — an insertion
make-ppd-of-ind.ind = ins

compose-ppd : ProPreDen x ProPreDen ⟼ ProPreDen
compose-ppd.(ppd-1, ppd-2) = ppd-1 ● ppd-2

Set-theoretically first two constructors are identity functions, but algebraically they "make" program preambles out of declarations and instructions. In the subsequent sections the constructors of DecDen and InsDen will be defined in such a way, that open-pro-den will not belong to their reachable parts.

---

[42] A temporary resignation from our denotation-to-syntax philosophy serves only an intuitive explanation of the structure of future programs.

6.4 Expressions

6.4.1 Value expressions

The denotations of *value expressions* are partial functions, that given a state return a value or an error:

ved : ValExpDen = WfState → ValueE value-expression denotations

We split value expressions into six categories. Contrary to the former case, all these categories belong to the common carrier of value expressions:

1. fixed-value expressions that return a typed data independently of the current state,
2. selection expressions that return a value pointed by a variable or an attribute of an object,
3. functional-procedure calls that return values built by procedures,
4. composed expressions associated with typed-data constructors.
5. boolean expressions,
6. conditional expressions.

The signatures of constructors of the denotations of value expressions are listed below.

Constructors of fixed-value-expression denotations

ved-bo.boo : ⟼ ValExpDen for boo : {tt, ff}
ved-in.int : ⟼ ValExpDen for int : Integer
ved-re.rea : ⟼ ValExpDen for rea : Real
ved-tx.tex : ⟼ ValExpDen for tex : Text

Constructors of selection-expression denotations

ved-variable : Identifier ⟼ ValExpDen
ved-attribute : Identifier x Identifier ⟼ ValExpDen

Constructor of functional procedure calls

ved-call-fun-pro : Identifier x Identifier x ActParDen ⟼ ValExpDen

Constructors based on typed-data constructors (examples)

ved-divide-re : ValExpDen x ValExpDen ⟼ ValExpDen
ved-create-li : ValExpDen ⟼ ValExpDen
ved-get-from-rc : ValExpDen x Identifier ⟼ ValExpDen
ved-get-from-ar : ValExpDen x ValExpDen ⟼ ValExpDen
…

Constructors of boolean-expression denotations

equal : ValExpDen x ValExpDen ⟼ ValExpDen
less : ValExpDen x ValExpDen ⟼ ValExpDen
ved-and : ValExpDen x ValExpDen ⟼ ValExpDen
ved-or : ValExpDen x ValExpDen ⟼ ValExpDen
ved-not : ValExpDen ⟼ ValExpDen

Conditional-expression constructor

ved-if : ValExpDen x ValExpDen x ValExpDen ⟼ ValExpDen

As we are going to see, fixed-value expressions always evaluate to typed data. The same will be true for composed expressions based on typed-data constructors. Consequently the only expressions that evaluate to objects will be selectors and functional-procedure calls. In the latter case procedure calls will return objects previously saved in stores and (possibly) modified by assigning new values to their attributes. Unlike in the case of typed-data expressions that return records, and may add or remove record attributes, object-oriented functional procedures may only modify the values assigned to attributes. This is an engineering decision.

The modifications of earlier created and declared objects are performed exclusively by instructions, and can only change values assigned to attributes. The only context where we can add a new attribute to an objecton are class declarations where we build class objectons (Sec. 6.7.4.2). These objectons are later used as patterns to build objects when we create object variables in stores by object constructors (Sec . 6.7).

All zero-argument constructors are defined accordingly to a common scheme which we show on the example of value expressions for integers. In this case we use a meta-constructor ved-integer which given an integer, e.g. 3 returns a zero-argument constructor in our algebra:

ved-int.3 : ⟼ ValExpDen i.e.
ved-int.3 : ⟼ WfState → Value | Error
ved-int.3.().sta =
  is-error.sta ➔ error.sta
  true ➔ (3, 'integer')

In this way, for every integer acceptable in our model we assume to have a dedicated constructor. Consequently, on the side of concrete syntax we can write constant-value expressions like 3, 245, or 340987502. If we had introduced only one zero-argument constructors corresponding to, say integer 1, then instead of writing 3 we had to write, e.g., ((1+1)+1).

The constructor that follows builds the denotations of expressions that return values assigned to state attributes, i.e. to variables:

ved-variable : Identifier ⟼ ValExpDen i.e.
ved-variable : Identifier ⟼ WfState ⟶ ValueE
ved-variable.ide.sta
  is-error.sta ➔ error.sta
  let
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
  obn.ide = ? ➔ 'variable not declared'
  dep.(obn.ide) = ? ➔ 'variable not initialized'
  true ➔ dep.(obn.ide)

The evaluation of a variable expression returns an error, if the variable hasn't been initialized[43].

Next constructor corresponds to an expression that returns a value assigned to an attribute of a computed object:

ved-attribute : ValExpDen x Identifier ⟼ ValExpDen i.e.
ved-attribute : ValExpDen x Identifier ⟼ WfState ⟶ ValueE
ved-attribute.(ved, ide).sta =
  is-error.sta ➔ error.sta
  ved.sta = ? ➔ ?
  ved.sta : Error ➔ ved.sta
  ved.sta /: Object ➔ 'object expected'
  let
    (obn, cl-ide) = ved.sta
  obn.ide = ? ➔ 'attribute unknown'
  let
    (tok, (typ, yok, ota)) = obn.ide the reference of ide in obn
    (env, (obn, dep, st-ota, sft, 'OK')) = sta
    dep.(obn.ide) = ? ➔ 'attribute not initialized'
    ota ≠ $ **and** ota ≠ st-ota ➔ 'attribute not visible'

[43] In their colloquial English programmers frequently say "a variable abc" when they mean (should say), "a variable expression abc". The difference between these concepts is clear at the level of abstract syntax where build-id.abc() is a variable and ved-expression(abc) is an expression. Then, at the level of colloquial syntax both are written as abc. which may lead to confusion.

true ➔ dep.(ob-obn.at-ide)

An expression that returns a value assigned to an attribute of an object may be evaluated successfully only if this attribute is visible in the current state. Here we realize the rules 1 and 2.1 of Sec. 5.4.3. Next constructor corresponds to calling a functional procedure:

call-fun-pro : Identifier x Identifier x ActParDen ⟼ValExpDen

We postpone its definition till Sec.6.6.4.2 where we discuss procedure calls.

As an example of a constructor based on a data constructor we show a constructor associated with the division of real numbers:

```
ved-divide-re: ValExpDen x ValExpDen ⟼ ValExpDen                    i.e.
ved-divide-re: ValExpDen x ValExpDen ⟼ WfState ⟶ Value | Error
ved-divide-re.(ved-1, ved-2).sta =
   is-error.sta          ➔ error.sta
   ved-i.sta = ?         ➔ ?              for i = 1,2
   ved-i.sta : Error     ➔ ved-i.sta      for i = 1,2
   let
      val-i  = ved-i.sta                  for i = 1,2
      val    = td-divide-rea.(val-1, val-2)
   true                  ➔ val
```

Our constructor "calls" the typed-data constructor td-divide-re (see Sec. 4.3) that performs the following actions:

1. checks if val-i's are of real types, and if val-2 is different from zero,
2. divides data parts of these values; this constructor also checks if the result is not too large, and if it is so, it generates an error message indicating an overflow,
3. returns the computed quotient or an error.

Since in the domain of typed data (Sec. 4.3) we have not defined constructors of boolean data (explanation below), we have to define constructors of boolean-expression denotations now, and we define them "from scratch". There are two groups of boolean expressions that we shall discuss. First group is built over comparison relations such as, e.g., an equality relation.

```
equal : ValExpDen x ValExpDen ⟼ ValExpDen                    i.e.
equal : ValExpDen x ValExpDen ⟼ WfState ⟶ Value | Error
equal.(ved-1, ved-2).sta =
   is-error.sta              ➔ error.sta
   ved-i.sta = ?             ➔ ?                 for i = 1,2
   ved-i.sta : Error         ➔ ved-i.sta         for i = 1,2
   let
      (cor-i, typ-i)  = ved-i.sta                for i = 1,2
   typ-1 ≠ typ-2             ➔ 'compared values must be of the same type'
   not comparable.typ-1      ➔ 'values not comparable'
   cor-1 = cor-2             ➔ (tt, 'boolean')
   true                      ➔ (ff, 'boolean')
```

We assume that comparable is a metapredicate (a parameter of our model) that distinguishes between comparable and not comparable values depending on their types. The use of this metapredicate explains why we have not introduced comparison constructors at the level of data[44].

---

[44] As a matter of fact, we could have introduced comparison constructors at the level of values, but for the sake of uniformity we decided to introduce them at the level of expressions where we define constructors corresponding to logical connectives.

Second group of boolean constructors is associated with logical connectives. Below we show an example of such a constructor associated with alternative:

```
ved-or : ValExpDen x ValExpDen ↦ ValExpDen                    i.e.
ved-or : ValExpDen x ValExpDen ↦ WfState → Value | Error
ved-or.(ved-1, ved-2).sta =
   is-error.sta           ➔ error.sta
   ved-1.sta = ?          ➔ ?
   ved-1.sta : Error      ➔ ved-1.sta
   let
      (cor-1, typ-1) = ved-1.sta
   typ-1 ≠ 'boolean'      ➔ 'boolean value expected'
   cor-1 = tt             ➔ (tt, 'boolean')
   ved-2.sta = ?          ➔ ?
   ved-2.sta : Error      ➔ ved-2.sta
   let
      (cor-2, typ-2) = ved-2.sta
   typ-2 ≠ 'boolean'      ➔ 'boolean value expected'
   cor-2 = tt             ➔ (tt, 'boolean')
   true                   ➔ (ff, 'boolean')
```

Note that the constructed expression denotation is not transparent for errors, and even not for undefinedness. If ved-1 evaluates to (tt, 'boolean'), then the final result is (tt, 'boolean') even if the evaluation of ved-2 generates an error or loops. This evaluation pattern is referred to as *lazy evaluation*. The opposite is an *eager evaluation* — as in all remaining examples of constructors — where we evaluate both subexpressions in the first place, and only then try to calculate the final result. Due to the laziness of our constructor an expression like

x > 0 **implies** 1/x > 0 i.e. x ≤ 0 **or** 1/x > 0

is true for all values of x that are less or equal zero. Note that with an eager evaluation it would generate an error for x = 0. Note, however, that at the same time a "nonsensical" expression

x > 0 **implies** x+y

is also true for x that are less or equal zero. In this case for x greater than zero our expression will generate an error message 'boolean value expected'.

Our last constructor correspond to if-then-else-fi expressions.

```
ved-if : ValExpDen x ValExpDen x ValExpDen ↦ ValExpDen

ved-if.(ved-1, ved-2, ved-3).sta =
   is-error.sta           ➔ error.sta
   ved-1.sta = ?          ➔ ?
   let
      val-1 = ved-1.sta
   val-1 : Error          ➔ val-1
   let
      (cor-1, typ-1) = val-1
   typ-1 ≠ 'boolean'      ➔ 'boolean-expected'
   cor-1 = tt             ➔ ved-2.sta
   cor-1 = ff             ➔ ved-3.sta
```

Here we also have to do with a lazy evaluation. In this case the advantage of laziness is even better visible. Consider the following example written in an anticipated syntax:

if x > 0 **then** sqr(x) **else** sqr(-x) **fi**

where sqr(x) denotes the square root of x. With an eager evaluation this expression evaluates to an error for all x except x = 0.

One methodological remark is necessary at the end and it concerns a question why we do not introduce in our algebra of denotations a carrier of boolean expressions, i.e., expressions with boolean values? We might use such expressions in building **if-then-else-fi** and **while-do-od** instructions, syntactically eliminating in this way the source of a typing errors, when a control expression generates a not-boolean value. Let us then analyze consequences of such a solution.

If we assume to have boolean expressions, then in seems natural to have among them boolean variables. The denotations of such variables would be then constructed by the following constructor:

```
boo-variable : Identifier ⟼ BooExpDen
boo-variable : Identifier ⟼ WfState ⟶ BooValE
boo-variable.ide.sta
   is-error.sta                        ➔ error.sta
   let
      (env, (obn, dep, st-ota, sft, ‘OK’)) = sta
   obn.ide = ?                         ➔ ‘variable not declared’
   dep.(obn.ide) = ?                   ➔ ‘variable not initialized’
   sort-va.(dep.(obn.ide)) ≠ ‘boolean’ ➔ ‘boolean value expected’
   true                                ➔ dep.(obn.ide)
```

This constructor checks if the value of the variable is boolean, whereas the earlier defined constructor

val-variable : Identifier ⟼ BooExpDen

does not check this property. Consequently at the level of abstract syntax (Sec. 7.2.3) we would have two categories or variables

ved-variable(ide) and
boo-variable(ide)

So far, everything is fine. But what about concrete syntaxes of variables? Can the abstract-to-concrete homomorphism glue them together? The answer is, not, since their denotations are different. The consequence of having boolean expressions as a syntactic category is therefore to have two syntactic categories of variables. Such a solution, although technically possible, seems not quite practical, since in such a case variables had to be somehow marked.

6.4.2 Yoke expressions

Yoke expressions in **Lingua** are used in the declarations of variables and in the declarations of class attributes (Sec. Sec. 6.7.4.2 and 6.7.4.3). We shall not use them in procedure declarations since the references of formal parameters will be getting the yokes of the references of actual parameters (Sec. 6.6.3.4). They are also extensively used in **Lingua-SQL** (Sec. 11), but technically in a differ way than here.

Since yokes will not be storable — an engineering decision to be seen in Sec. 5.3 — the denotations of yoke expressions could have been made just yokes. Nevertheless, we define them as functions on states to allow for the generation of values — used in the creation of yokes — by value expressions.

yed : YokExpDen = WfState → YokeE

Consequently, we have to assume that our denotations are partial functions and that they me return errors instead of yokes. In this way yoke-expression denotations may generate errors on two levels: when they generate yokes, and when the generated yokes are applied to values.

Constructors of yoke-expression denotations are derived from constructors of yokes defined in Sec. 4.4. In the signatures of these constructors domain Yoke is replaced by YokExpDen and domain TypDat by ValExpDen. E.g., from yoke constructor

yo-give-td : TypDat ⟼ Yoke

we derive the following constructor of denotations

```
yed-give-td : ValExpDen ⟼ YokExpDen          i.e.
yed-give-td : ValExpDen ⟼ WfState → YokeE
yed-give-td.ved.sta =
   is-error.sta              ➔ error.sta
   ved.sta = ?               ➔ ?
   let
      val = ved.sta
   val : Error               ➔ val
   sort-t.val : Identifier   ➔ 'objects are not allowed'
   true                      ➔ yo-give-td.val
```

This constructor given a value-expression denotation returns a yoke-expression denotation that given a state generates a yoke that given an arbitrary typed data generates the value of the value expression. The generated yoke is a partial function since it "calls" a value-expression denotation that is partial.

Another example of a constructor that takes a value-expression denotation as an argument is the constructor corresponding to the yoke that gets an element of an array:

```
yed-get-from-ar : ValExpDen ⟼ YokExpDen
yed-get-from-ar : ValExpDen ⟼ WfState → YokeE
yed-get-from-ar.ved.sta =
   is-error.sta              ➔ error.sta
   ved.sta = ?               ➔ ?
   let
      val = ved.sta
   val : Error               ➔ val
   sort-t.val ≠ 'integer'    ➔ 'integer expected'
   true                      ➔ yo-get-from-ar.val
```

The definitions of the remaining constructors are analogous.

6.4.3 Type expressions

Type expressions are used in four contexts:

1. in type declarations, where we build a new type and store it in a type environment of a class for future use,
2. in variable declarations, where we declare a new variable and assign a profile to it (we recall that types are components of profiles),
3. in attribute declaration — analogously,
4. in pre-procedure declarations, where we assign profiles to formal parameters.

The signatures of constructors of type-expressions denotations are the following:

```
ted-create-bo    :                                     ⟼ TypExpDen
ted-create-in    :                                     ⟼ TypExpDen
ted-create-re    :                                     ⟼ TypExpDen
ted-create-tx    :                                     ⟼ TypExpDen
ted-create-ot    : Identifier                          ⟼ TypExpDen

ted-constant     : Identifier x Identifier             ⟼ TypExpDen
ted-create-li    : TypExpDen                           ⟼ TypExpDen
ted-create-ar    : TypExpDen                           ⟼ TypExpDen
ted-create-re    : Identifier x TypExpDen              ⟼ TypExpDen
ted-put-to-re    : Identifier x TypExpDen x TypExpDen  ⟼ TypExpDen
```

First constructor is a zero-argument constructor that creates a boolean-type expression denotation "out of nothing":

```
ted-create-bo.().sta =
   is-error.sta   ➔ error.sta
   true           ➔ 'boolean'
```

The presence and the role of this constructor in **AlgDen** is the same as in the case of the constructors of identifiers. The remaining zero-argument constructors are defined in a similar way.

Constructors of the next subgroup are created from these type constructors that create "new types". For instance:

```
ted-create-re.(ide, ted).sta =
   is-error.sta       ➔ error.sta
   ted.sta : Error    ➔ ted.sta
   let
      typ = ted.sta
   true               ➔ ty-create-re.(ide, typ)
```

This constructor calls constructor ty-create-re from our algebra of typed data. Note that, e.g., the body constructor ty-put-to-re does not create a new body, and therefore we do not introduce a corresponding constructor of denotations.

So far we have defined constructors corresponding to the types of data. Our next constructor builds the denotation of an expression that returns a type of an object, i.e. an identifier:

```
ted-create-ot.ide.sta =
   is-error.sta   ➔ error.sta
   true           ➔ ide
```

Although intentionally ide is supposed to be the name of a class, we do not check, if this is indeed the case, since — as we are going to see in Sec. 6.7.4.2 — we may need to define an object type (temporarily) associated with a class which "hasn't been declared yet". However, as we are going to see in Sec. 6.7.4.3, such "undefined object types", will not be declarable.

Our last constructor is ted-constant that corresponds to a type-constant. This constructor describes the action of reading a previously declared type from a type environment of a class:

```
ted-constant.(ide-cl, ide-ty).sta =
   is-error.sta          ➔ error.sta
   let
      ((cle, mee, cov), sto) = sta
   cle. ide-cl = ?       ➔ 'class unknown'
   let
      (ide-cl, tye, mee, obn) = cle.ide-cl                      well-formedness of sta
   tye.ide-ty = ?        ➔ 'type unknown'
   tye.ide-ty = Θ        ➔ 'type not concretized'
   true                  ➔ tye.ide-ty
```

We are talking here about "constants", since a type once assigned to an identifier in a class can't be changed.

At the end a comment about structured data types such as list-, array, or record types. Notice that object types are never structural in this way, i.e. we may build a type of integer arrays, but not of object arrays. This is an engineering decision.

6.4.4 Reference expressions

In programming languages without such deep value-structures such as our objects, a reference on the left-hand side of an assignment is represented by a single identifier. In our case the situation is different, since we may wish to assign a value to a deep reference in an object, as, e.g.,

```
node.next.no := 11
```

(cf. example discussed in Sec. 5.1). To handle this problem, we introduce expressions that given a state return a reference, or an error:

red : RefExpDen = WfState ⟼ ReferenceE

We shall need only two constructors of the denotations of such expressions. The first one corresponds to a single variable:

```
ref-variable : Identifier ⟼ RefExpDen                    i.e.
ref-variable : Identifier ⟼ WfState ⟼ ReferenceE
ref-variable.ide.sta =
   is-error.sta   ➔ error.sta
   let
      (env, (obn, dep, st-ota, sft, ‘OK’)) = sta
   obn.ide = ?   ➔ ‘variable not declared’
   true          ➔ obn.ide
```

Our second constructor builds expression denotations that may return references assigned to deep attributes of objects:

```
ref-attribute : ValExpDen x Identifier ⟼ RefExpDen                    i.e.
ref-attribute : ValExpDen x Identifier ⟼ WfState ⟼ ReferenceE
ref-attribute.(ved, at-ide).sta =
   is-error.sta                      ➔ error.sta
   ved.sta = ?                       ➔ ?
   ved.sta : Error                   ➔ ved.sta
   ved.sta /: Object                 ➔ ‘object expected’
   let
      (va-obn, va-ide)                    = ved.sta
      (env, (obn, dep, st-ota, sft, ‘OK’)) = sta
   va-obn.at-ide = ?                 ➔ ‘attribute not declared’
   let
      (tok, (typ, yok, at-ota)) = va-obn.at-ide
   at-ota ≠ $ and at-ota ≠ st-ota    ➔ ‘attribute not visible’
   true                              ➔ va-obn.at-ide
```

Here we realize the rules 1. and 2.2 of Sec. 5.4.3. As we see, references computed by reference expressions are always pointed either by variables or by object attributes. In other words, the only operations that allow us to get references are selection operations. It is, of course, an engineering decision.

6.5 Instructions

6.5.1 Signatures of constructors

Instructions modify state by assigning new values to variables and attributes. We shall define the following constructors of instruction denotations:

atomic instructions

assign : RefExpDen x ValExpDen ⟼ InsDen assignments

call-imp-pro : Identifier x Identifier x ActParDen x ActParDen ⟼ InsDen imp. proc. calls
call-obj-con : Identifier x Identifier x Identifier x ActParDen ⟼ InsDen obj. const. calls
skip-ins : ⟼ InsDen trivial instruction

structural instructions

if : ValExpDen x InsDen x InsDen ⟼ InsDen conditional instructions
if-error : ValExpDen x InsDen ⟼ InsDen error elaboration
while : ValExpDen x InsDen ⟼ InsDen while loops
compose-ins : InsDen x InsDen ⟼ InsDen sequential compos.

Atomic instructions are called in this way, since they do not include other instructions as their components.

The skip instruction has a technical character and has been introduced to cover the case of functional procedures (Sec.6.6.4.1) whose bodies consist of an expression alone, i.e. without a preceding program. Its denotation is an identity function on states.

The calls of imperative procedures and of object constructors will be described in Sec. 6.6.3.6 and Sec. 6.6.5.3.

6.5.2 Assignment instructions

An *assignment instruction* computes a reference and a value, and then assigns this value to this reference in the current deposit:

```
assign : RefExpDen x ValExpDen ⟼ InsDen
assign : RefExpDen x ValExpDen ⟼ WfState → WfState
assign.(red, ved).sta =
   is-error.sta                    ➔ error.sta
   ved.sta = ?                     ➔ ?
   ved.sta : Error                 ➔ sta ◄ ved.sta
   red.sta  : Error                ➔ sta ◄ red.sta
   let
      val                          = ved.sta
      ref                          = red.sta
      (tok, (typ, yok, re-ota))    = ref
      (env, (obn, dep, st-ota, sft, ‘OK’))  = sta
   re-ota ≠ $ and re-ota ≠ st-ota  ➔ sta ◄ ‘reference not visible’
   not ref VRA.cov val             ➔ sta ◄ ‘incompatibility of types’
   yok.val : Error                 ➔ sta ◄ yok.val
   sort.(yok.val) ≠ ‘boolean’      ➔ sta ◄ ‘yoke not boolean’
   yok.val = (ff, ‘boolean’)       ➔ sta ◄ ‘yoke not satisfied’
   let
      new-sta = (env, (obn, dep[ref/val], st-ota, sft, 'OK'))
   true                            ➔ new-sta
```

In this definition we realize the rule 2.2 of Sec. 5.4.3.

6.5.3 Structural instructions

Structural instructions are built from atomic instructions using four constructors announced in Sec. 6.4. A conditional composition of instructions is defined as follows:

```
if : ValExpDen x InsDen x InsDen ⟼ InsDen

if.(ved, ind-1, ind-2).sta =
   is-error.sta        ➔ sta
   ved.sta = ?         ➔ ?
```

```
    ved.sta : Error       ➔ sta ◄ ved.sta
    let
       val = ved.sta
    val : Object          ➔ sta ◄ 'typed data expected'
    let
       (dat, typ) = val
    typ ≠ 'boolean'       ➔ sta ◄ 'boolean value expected'
    dat = tt              ➔ ind-1.sta
    true                  ➔ ind-2.sta
```

Note that due to while loops (see below) and imperative-procedure calls (Sec. 6.6.3.6) the execution of both component instructions may be infinite, which means that the state ind-1.sta or ind-2.sta may be undefined.

The next structural constructor is related to an *error-handling mechanism*. It activates a *rescue action* that is an instruction associated with an error message indicated by value expression, called *error trap*, whose value is a word identical with this message.

if-error : ValExpDen x InsDen ⟼ InsDen

```
if-error.(ved, ind).sta =
    not is-error.sta  ➔ sta
    let
       message  = error.sta
       sta-1    = sta ◄ 'OK'
    ved.sta-1 = ?     ➔ ?
    let
       val = ved.sta-1
    val : Error       ➔ sta ◄ 'trap generates an error'
    let
       (cor, typ) = val
    typ ≠ 'text'      ➔ sta ◄ 'word expected'
    cor ≠ message     ➔ sta ◄ 'trap not adequate'
    ind.sta-1 = ?     ➔ ?
    let
       sta-2 = ind.sta-1
    is-error.sta-2    ➔ sta ◄ 'rescue action generates an error'
    true              ➔ sta-2
```

If the input-state sta does not carry an error, then this state becomes the output state, since there is no error to handle.

In the opposite case, a temporary state sta-1 is created by removing error err from sta. In the new state, we compute the value of the trap expression ved. Seven situations may happen in this moment:

1. the evaluation of trap expression does not terminate,
2. the evaluation terminates, but the computed value is an error,
3. the computed value is not a word value,
4. the computed value is a word value, but its data part is different from the error message that we want to trap,
5. the computed value caries the trapped message, but the rescue instruction does not terminate ,
6. the rescue instruction terminates but it generates an error message itself,
7. the rescue instruction terminates without an error, and its terminal state is the resulting state.

Of course, the above constructor should be regarded as an example, only showing that error-handling mechanisms may be described in our model.

The definition of the constructor of **while** loops involves a fixed-point definition of the constructed instruction:

while : ValExpDen x InsDen ⟼ InsDen

while.(ved, ind).sta =
  is-error.sta ➔ sta
  ved.sta = ? ➔ ?
  ved.sta : Error ➔ sta ◄ ved.sta
  let
    val = ved.sta
  val : Object ➔ 'typed data expected'
  let
    (dat, typ) = val
  typ ≠ 'boolean' ➔ sta ◄ 'boolean value expected'
  dat = ff ➔ sta
  ind.sta = ? ➔ ?
  ind.sta : Error ➔ ind.sta
  true ➔ (while.(ved, ind))).(ind.sta)

Notice that the unique (least) solution of this equation is not the while constructor, but the effect of its application to its arguments, i.e. while.(ved, ind).

Due to **while** instructions, the denotations of instructions may be partial functions. The partiality of while.(ved, ind) may happen in three situations:

1. the evaluation of the boolean expression ved does not terminate; this may be the case if ved calls a functional procedure,
2. the execution of the body ind does not terminate,
3. the execution of the "main loop" does not terminate.

Comment 6.5.3-1 In the definition of **while** we have to do with a fixed-point equation in a CPO of partial functions **InsDen** (Sec. 2.7). For any pair **(ved, ind)** the solution of this equation is the denotation:

while.(ved, ind) : State → State

To see this equation written explicitly in our CPO, let us introduce the following notations:

NotOK = {(sta, sta) | is-error.sta}
ExpEr = {(sta, sta ◄ved.sta) | ved.sta : Error}
IsObj = {(sta, sta ◄'typed data expected) | ved.sta : Object}
NotBoo = {(sta, sta ◄'boolean-expected') | ved.sta is not a boolean value}
FF = {(sta, sta) | ved.sta = (ff, 'boolean')}
TT = {(sta, sta) | ved.sta = (tt, 'boolean')}

Now, our equation is the following:

X = NotOK | ExpEr | IsObj | NotBoo | FF | TT•ind•X

Since the operators | and • are continuous, the least solution of that equation exists, and since the coefficients of that equations have mutually disjoint domains, from Theorem 2.7-1 we may conclude that its solution is a function, and may be described by the following formula:

X = (TT• ind)* • (NotOK | ExpEr | IsObj | NotBoo | FF)

The last constructor of structural instructions corresponds to sequential composition of instructions, and is the following:

compose-ins : InsDen x InsDen ⟼ InsDen
compose-ins.(ind-1,ind-2) = ind-1 ● ind-2

Sequentially composed instructions are executed one after another.

6.6 Methods

6.6.1 An overview of methods

Methods in our model fall into three basic operational categories:

- imperative methods,
- object constructor methods,
- functional methods.

In each of these categories a method may be *abstract* or *concrete*. Abstract methods will be otherwise called *procedure signatures*, and concrete methods — *procedures* or *object constructors* respectively. The domain of methods and related domains are defined by the following equations (for actual-parameter denotations and the denotations of signature (see Sec. 6.1):

met	: Method	= Procedure | ProSigDen		methods
pro	: Procedure	= ImpPro | FunPro | ObjCon		procedures
ipr	: ImpPro	= ActParDen x ActParDen	$\mapsto$ Store $\to$ Store	imperative procedures
fpr	: FunPro	= ActParDen x TypExpDen	$\mapsto$ Store $\to$ ValueE	functional procedures
oco	: ObjCon	= ActParDen x Identifier	$\mapsto$ Store $\to$ Store	object constructors[45]
prs	: ProSigDen	= ImpProSigDen | FunProSigDen | ObjConSigDen		proc. signature denotations

It must be emphasized that domains associated with concrete methods are not the carriers of our algebra of denotations (cf. Sec. 6.1). Therefore, they will not have syntactic counterparts. This is why we are not talking about “procedure denotations”, but about “procedures” as such. At the side of syntax we will only have procedure declarations and calls, and their denotation will belong to the denotations of declarations and of instructions respectively.

In turn, procedure signatures will be represented at the side of syntax, and therefore we talk about their denotations. Their simple constructors will be defined in Sec. 6.6.2.

It may be worth mentioning in this place that procedures and procedure-signature denotations belong to two different worlds. Procedures are functions that given actual parameters return store-to-store functions. Note that actual parameters do not have types, since they are just identifiers. In turn, signatures are not functions. They are lists of formal-parameter denotations, and formal parameter do have types! Signatures may be said to be “incomplete procedure-declarations” (they have no bodies), whereas procedures are the effects of procedure declarations.

In this overview we shall concentrate on procedures, since their model requires some specific solutions.

Let’s start from imperative procedures that are functions which take two lists of *actual parameters* — *value parameters* and *reference parameters* — and return store-to-store functions. We shall assume that reference parameters will constitute a unique communication channel between procedures and the “external word”. A possible alternative might be the introduction of global variables, but such a solution would complicate rules of the construction of correct procedure calls (cf. Sec. 9.4.6.3), and besides would be — in our opinion —error prone. We want to make sure that all interactions of a procedures with global states will be explicit in their declarations.

A second important issue about procedures is that they modify stores rather than states. Although procedure calls will be state-to-state functions, we can’t assume procedures to be such functions, since it would

[45] “Object constructors” should not be confused with “constructors in algebras”. The former constitute a category of procedures, whereas the latter are functions “between” carriers of algebras. We decided to use the same word in both cases because the literature has already established customs to call them that way.

lead to a situation where a procedure may take as an argument a state, where this procedure has been declared. In a simplified version such a situation would lead to the following set of domain equations:

Procedure = State → State
State = Identifier ⇒ Procedure

This set can't be solved on the ground of usual set theory, since no function can take itself as an argument. Self-applicable functions constitute so called *reflexive domains*[46], and have been used in early denotational models of Algol 60, where a procedure can take itself as a procedural parameter[47].

After these explanations, let's note that to use procedures in a programming language we need tolls:

- to create them,
- to declare them, i.e. to save them in states,
- to call, and execute them.

In earlier versions of our denotational model investigated in [30], [32], [35] and [39] procedures were created by declarations out of procedures' components and were saved in the environments of states. Recursive procedures were defined as least solutions of single fixed-point equations and mutually recursive procedures as least solutions of sets of such equations.

If we would like to apply this mechanism in our case, we had to assume that all procedure declarations of all classes are elaborated simultaneously, meaning that all classes have to be declared simultaneously and recursively. Such a solution would lead to technical complications since, in that case, we would have to define a chain-complete partial order (a CPO, see Sec. 2.4) in the domain of classes. A CPO in a set of tuples of a common length (like classes) is usually defined componentwise. Now, whereas to define a CPO in the domain of procedure environments is a rather routine task, it is not clear (at least not clear for us), how to define an adequate CPO in the domains of type environments and objectons.

Facing this problem we decided to move the creation of procedures from the time of their declarations in classes to a later time when all classes have been declared, but prior to the time when procedures are called. Technically the declarations of procedures in classes will not create and store procedures, but only *pre-procedures* that will be later used to create all procedures "in one step" once all classes have been declared. In this step programs perform one global declaration open-pro-den (Sec. 6.7.6). To simplify future rules of program construction we have assumed in Sec. 6.3 that this operation appears only once in every program and is located between declarations and instructions.

As a consequence of our assumption, pre-procedures will be defined as functions that given an environment return a procedure. The domains of pre-procedures are therefore the following:

ppr : PrePro	= ImpPrePro | FunPrePro | ObjPreCon	pre-procedures
ipp : ImpPrePro	= Env ⟼ ImpPro	imperative pre-procedures
fpp : FunPrePro	= Env ⟼ FunPro	functional pre-procedures
opc : ObjPreCon	= Env ⟼ ObjCon	object pre-constructors

When a procedure pro is called we execute the corresponding pre-pro in a declaration time environment dt-env, i.e., we execute the function

pre-pro.dt-env : Store → Store

which given a call-time store returns a new store. The declaration-time environment is common to all procedures, and is the environment passed to open-procedures. Since no declarations follow open-procedures, all states that follow the execution of this declaration have a common environment which differs from the declaration-time environment dt-env only by having procedures declared in procedure environment pre.

---

[46] A model of self-applicable functions has been described by Dana Scott and Christopher Strachey [84] in 1970., but its technical complexity discouraged researchers from its use. Independently it turned out that the use of self-applicable functions in programming may be error prone.

[47] This mechanism was implemented in Algol 60 by the so-called "copy rule", where a compiler or interpreter copied the text of a procedure body into the context of a program where this procedure was to be used.

6.6.2 Signatures and parameters

We start from two simple constructors of lists (tuples) of identifiers:

build-loi : Identifier ⟼ ListOfIde
add-to-loi : Identifier x ListOfIde ⟼ ListOfIde

We skip their obvious definitions. Given this domain we may define *declaration sections* that consist of lists of identifiers followed by a type-expression denotation and a yoke-expression denotation :

build-dse : ListOfIde x TypExpDen ⟼ DecSec

Such a section expresses the fact that the given identifiers are formal parameters of a given type, e.g., at the side of syntax:

x, y, z array-of-integers

Formal-parameter denotations are tuples of declaration sections, hence we need two constructors to build their domain:

build-fpd : DecSec ⟼ ForParDen
add-to-fpd : DecSec x ForParDen ⟼ ForParDen

First constructor makes a formal parameter denotation out of a declaration section, the second — adds a new section to a parameter denotation. Their definitions are obvious. Now, we can show signatures of three constructors of the domains of formal and respectively actual parameters:

build-ipsd : ForParDen x ForParDen ⟼ ImpProSigDen
build-fpsd : ForParDen x TypExpDen ⟼ FunProSigDen
build-ocsd : ForParDen x ForParDen ⟼ ObjConSigDen

build-apd : ListOfIde ⟼ ActParDen build actual-parameter denotations

We again skip their obvious definitions.

At the end let us explain the idea of abstract methods, i.e. of signatures. As we know, an important part of classes constitute methods. Ultimately these methods should be concrete, since only then we may use (call) them. However, we may wish to define a parent class with abstract methods to have more flexibility in the creation of their (inheriting) children classes. We thus do not define concrete procedures, but only their "types" represented by lists of parameters.

6.6.3 Imperative pre-procedures

6.6.3.1 An intuitive understanding

First step on the way to understand the mechanism of imperative procedures is to understand the execution of their calls. Since an imperative-procedure call is an instruction, it takes an initial state, and transforms it into a terminal state. These states will be called *global states* to distinguish them from *local states* that the procedure creates to execute its body.

The execution of a call of a procedure declared in class MyClass is performed in four steps illustrated in Fig. 6.6-1.

1. A *local initial store* li-sto is created where initially the only variables bound in the objecton are the identifiers of formal parameters. Value parameters point to new references, and these references point to the twins (Sec. 4.4) of the values of actual value-parameters. Reference parameters point to the references of actual parameters. Local initial deposit carries all reference of the global deposit, but all these references, except the references of reference parameters, are orphan references. The origin tag of the local store is MyClass.

2. A *local initial state* is created by combining a *declaration-time environment* dt-env with the local initial store. The declaration-time environment must be, therefore, somehow "remembered" when the procedure is declared.
3. The *local initial state* is transformed by the body of the procedure (a deep program) into a *local terminal state* (lt-env, lt-sto). During the execution of the body some local (temporary) value variables, classes, types and procedures may be declared.
4. The local terminal state is transformed into a *global terminal state*, with the (unchanged) *global initial environment* gi-env and a *global-terminal store* gt-sto, where actual reference parameters regain access to their earlier references. All locally declared items cease to exist.

One comment is necessary about the declaration-time environment mentioned in point 2. As we are going to see in Sec. 6.7.6, this environment is "loaded" to a procedure when this procedure is created by a global declaration open-procedures, which is executed as the last declaration in a program (Sec. 6.3). Consequently, the environments of all consecutive states of the program have the same environment which is the output environment of global declaration.

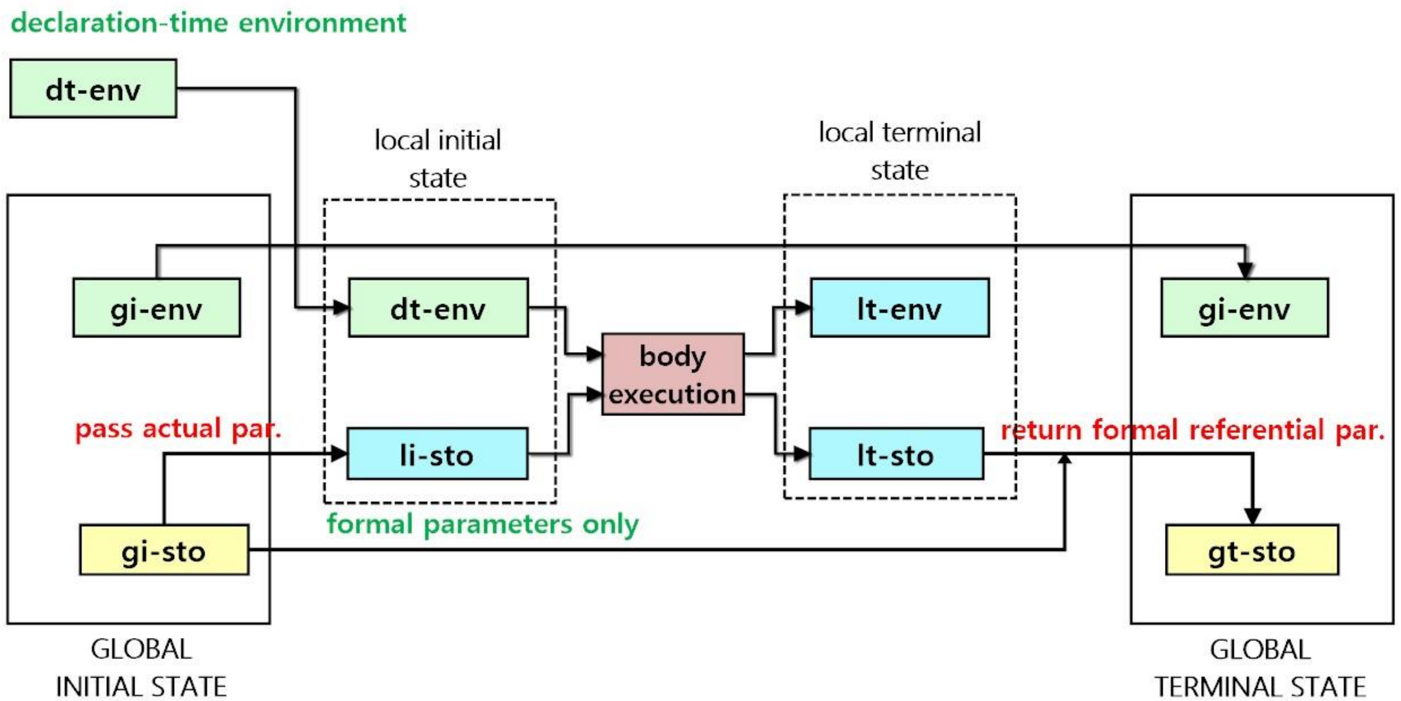


Fig. 6.6-1 The execution of an imperative-procedure call

6.6.3.2 Creating imperative pre-procedures

A formal description of the execution illustrated in Fig. 6.6-1 is included in the definition of a constructor of imperative pre-procedures. We recall that pre-procedures became procedures when they receive an environment as an argument. In the definition below we use two functions describing the mechanisms of passing and returning parameters, which will be defined later in Sec.6.6.3.4 and Sec.6.6.3.5 respectively.

```
create-imp-pre-pro : ImpProSigDen x ProDen x Identifier ⟼ ImpPrePro
create-imp-pre-pro : ForParDen x ForParDen x ProDen x Identifier ⟼
                                        ⟼ Env ⟼ ActParDen x ActParDen ⟼ Store → Store
create-imp-pre-pro.(fpd-v, fpd-r, prd, cl-ide).dt-env.(apd-v, apd-r).ct-sto =
   is-error.ct-sto     ➔ ct-sto                                          dt- declaration time
   let                                                                   ct- call time
      li-sto = pass-actual.(fpd-v, fpd-r, apd-v, apd-r, cl-ide).dt-env.ct-sto     li-  local initial
   is-error.li-sto     ➔ ct-sto ◄ error.li-sto
   let
      li-sta = (dt-env, li-sto)                                             local initial state
```

```
prd.li-sta = ?          ➔ ?
let
   lt-sta = prd.li-sta                                              local terminal state
is-error.lt-sta         ➔ ct-sto ◄ error.lt-sta
let
   (dt-cle, dt-pre, dt-cov)   = dt-env
   (lt-env, lt-sto)           = lt-sta
   gt-sto                     = return-formal.fpd-r.ct-sto.lt-sto.dt-cov
is-error.gt-sto         ➔ ct-sto ◄ error.gt-sto
true                    ➔ gt-sto                                    global terminal store
```

First, the pre-procedure builds a *local initial store* by passing actual parameters to formal parameters and by setting cl-ide, the name of a class, as the origin tag of the store. As we will see in Sec. 6.7.4.6, cl-ide will be the class name where our pre-procedure will be defined. Then, it creates a *local initial state* by combining the declaration-time environment with the call-time store.

The local initial state is transformed into a local terminal state by a program prd that constitutes the body of the procedure.

If this transformation terminates, and does not issue an error, then the global terminal store is created by returning the references of formal reference parameters to actual reference parameters, and by going back to the call-time origin tag. If this store carries no error message, then it is issued by the procedure. Next the mechanism of procedure call (Sec. 6.6.3.6 and Fig. 6.6-1) combines the global-terminal store with call-time environment into global-terminal state.

6.6.3.3 A static compatibility of parameters

The first step in passing actual parameters to formal parameters consist in checking if they are statically compatible with each other, i.e., if the lists of corresponding identifiers are of the same length, and additionally there are no repetitions of identifiers on the list of formal parameters. To formalize this checking process we define two auxiliary functions. We skip their (rather obvious, but tedious) formal definitions showing only examples:

```
list-of-for-par : ForParDen ↦ (Identifier x TypExpDen)^c*              e.g.
list-of-for-par.(((x, y, z), ted-1), ((q, r), ted-2)) =
   ((x, ted-1), (y, ted-1), (z, ted-1), (q, ted-2), (r, ted-2))

list-of-ide : (Identifier x TypExpDen)^c*  ↦ Identifier^c*               e.g.
list-of-ide.((x, ted-1), (y, ted-1), (z, ted-1), (q, ted-2), (r, ted-2)) = (x, y, z, q, r)
```

Now, the checking function is defined as follows:

```
statically-compatible : ForParDen x ForParDen x ActParDen x ActParDen ↦ Error | {'OK'}

statically-compatible.(fpd-v, fpd-r, apd-v, apd-r) =
   let
      for-val-par = list-of-ide.(list-of-for-par.fpd-v))
      for-ref-par = list-of-ide.(list-of-for-par.fpd-r))
      for-par = for-val-par © for-ref-par
   are-repetitions.for-par               ➔ 'formal par repetitions'
   length.for-val-par ≠ length.apd-v     ➔ 'incompatible numbers of value parameters'
   length.for-ref-par ≠ length.apd-r     ➔ 'incompatible numbers of reference parameters'
   true                                  ➔ 'OK'
```

In words, the lists of formal and actual parameter denotations of a procedure call are statically compatible if:

1. no formal parameter appears twice on a combined list of value- and reference parameters; a similar property of actual value-parameters is, of course, not required,
2. the mutually corresponding lists of formal and actual parameter denotations are of the same lengths.

Note that empty lists of corresponding parameters are compatible.

The defined property is called *static* since it can be checked at compilation time, i.e., before program execution. Note that "static" does not mean "syntactic" — static compatibility requirement can't be built into a grammar, and therefore it can't be checked by a syntax analyser.

6.6.3.4 Passing actual parameters to a procedure

Ten rozdział przeczytajcie szczególnie uważnie, bo jest w nim wiele technikaliów. ???

Function pass-actual describes the process of passing the values and references of actual parameters of a procedure call to the body of the called procedure. Technically it creates a local initial store to be elaborated by the body (Sec.6.6.3.2). Actual value-parameters must be declared and initialized, and actual reference-parameters must be declared, but not necessarily initialized.

Fig. 6.6-2 illustrates the mechanism of passing parameters. As was already shown in Sec. 6.6.3.2, the environment of the local input state will be a declaration-time environment whereas its store will be a local initial store created by our function.

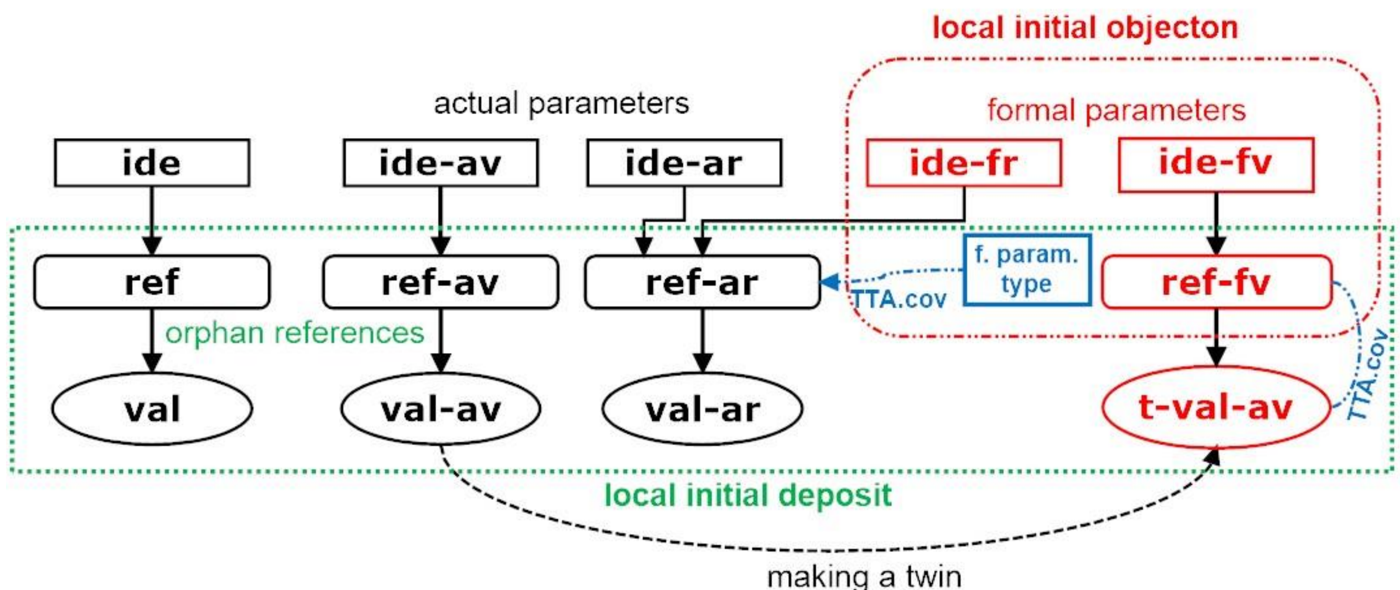


Fig. 6.6-2 Passing actual parameters to a procedure (a simplified picture)

We assume the following rules about the creation of local initial stores. All of them have an engineering character:

1. The only identifiers bound initially in the local store are formal parameters. Of course, during the execution of the call some local variables may be added (declared). As a consequence, procedures can't use global variables and, therefore, their only "side effects" are due to reference parameters.
2. The current references of actual reference-parameters ref-ar become the references of formal reference parameters ide-fr. To "meet the expectations" of procedure's designer, the actual types of ide-ar's, i.e., the types of ref-ar's, must be accepted by the declared types of formal parameters ide-fr's. In turn formal reference-parameters receive the yokes of actual parameters. We recall that when we declare a procedure, we indicate the types of its formal parameters but leave their yokes unspecified (a mathematical decision). Otherwise, we had to ensure the compatibility of actual with formal yokes, i.e., we had to compare yokes, which might be challenging to implement.
3. Fresh references ref-fv's are created for formal value-parameters. The types of these references are the declared types of formal parameters, and their yokes are the yokes of actual references ref-av's. To these references we assign the twins (see later) t-val-av's of the values val-av's of actual parame-

ters. The new references must accept these twins. The picture shown in Fig. 6.6-2 is "simplified" since making a twin of an object requires replacing all tokens of its references by fresh ones, which may result in substantial modification (enrichment) of the deposit.

4. The origin tags of actual parameters become the origin tags of formal references, which seems to be an obvious choice. If an actual parameter is public, the corresponding formal parameter should be public as well. In turn, if it is private, then it should remain private with the unchanged origin in the local state.

To define a function of passing actual to formal parameters, we shall need a function that given a value returns its *twin*. A twin of a typed data is just this data, and a twin of an object is created by replacing all tokens in this object by fresh ones. Before we proceed to a formal definition of this function let's observe the following facts:

1. A description of an object requires the context of a deposit, and therefore our function must modify deposits.
2. The replacement of tokens by fresh tokens consumes tokens, and therefore our function must modify sets of free tokens.
3. Due to the fact that objects may include cycles, we have to stop the replacement of "old" tokens by new ones whenever we encounter a token that "is already new". To do this our function will "monitor" a set of new tokens.

To define the twining function we shall need two new domains:

```
snt  : SetNewTok = Set.Token                     sets of new tokens[48]
tot  : TupleOfTok = Token^c*                     tuples of tokens
```

and two new functions:

```
tokens-of     : Value x Deposit                  ⟼ Set.Token
get-new-tok   : Integer x SetNewTok x SetFreTok  ⟼ TupleOfTok x SetNewTok x SetFreTok
```

The first function given a value and a deposit returns the set of all tokens included in this value. We skip its obvious definition. The second generates a tuple of fresh tokens and appropriately modifies the sets of new tokens and free tokens. Its definition is the following:

```
get-new-tok.(n, snt, sft) =
   n ≤ 0     ➔ 'number of tokens must be positive'
   n = 1     ➔
      let
         (tok, sft-1)     = get-tok.sft
         snt-1            = snt | {tok}
      ((tok), snt-1, sft-1)
   n > 1     ➔
      let
         (tok, sft-1)           = get-tok.sft
         snt-1                  = snt | {tok}
         (tot, snt-2, sft-2)    = get-new-tok.(n-1, snt-1, sft-1)
      ((tok) © tot, snt-2, sft-2)
```

The function that creates twins of values is the following:

```
create-twin : Value x Deposit x SetNewTok x SetFreTok ⟼
                                   ⟼ (Value x Deposit x SetNewTok x SetFreTok)
create-twin. (val, dep, snt, sft) =
   val : TypDat              ➔ (val, dep, snt, sft)
```

---

48 Formally speaking we do not introduce here a new domain since SetNewTok = Set.Token = SetFreTok, but a new metaname to distinguish between two arguments of our function **snt** and **sft** that are of the same sort.

```
    tokens-of.val ⊆ snt       ➔ (val, dep, snt, sft)
    let
       (obn, cl-ide)                              = val              the argument value is an object
       [ide-1/ref-1,…,ide-n/ref-n]                = obn
       (tok-i, prf-i)                             = ref-i
       ((new-tok-1,…,new-tok-n), sft-1, snt-1)    = get-new-tok.(n, snt, sft)
       new-ref-i  =
          tok-i : snt         ➔ ref-i
          true                ➔ (new-tok-i, prf-i)                        for i = 1;n
       obn-1                                      = [ide-1/new-ref-1,…,ide-n/new-ref-n]
       val-i                                      = dep.ref-i      for i = 1;n
       (twin-val-1, dep-1, snt-2, sft-2)          = create-twin.(val-1, dep, snt-1, sft-1)
       …
       (twin-val-n, dep-n, snt-(n+1), sft-(n+1))  = create-twin.(val-n, dep-(n-1), snt-n, sft-n)
       new-dep                                  = dep-n[new-ref-1/twin-val-1,…,new-ref-n/twin-val-n]
       twin-val                                   = (obn-1, cl-ide)
    true                      ➔ (twin-val, dep-n, , snt-(n+1), sft-(n+1))
```

In the first step our function checks if the value val to be "twinned" is a typed data, and if this is the case, it returns the same value. Deposit and both sets of tokens remain unchanged and the process stops.

If the value is an object (obn, cl-ide), then we check if all tokens in this value have been already replaced, and if this is the case then the process stops.

Otherwise our function replaces all not-new references in obn by fresh references, and appropriately modifies the sets of new tokens and free tokens. Of course, in the first step of our recursion none of these references are new, but in further steps such a situation may happen, if there are cycles in our object.

After the first step of our recursion we have a new objecton obn-1, but all its references are dangling. Then, for every val-i assigned to a reference of the original obn we create recursively a twin of this value, and we assign this twin to the corresponding reference of obn-1. In each such step we appropriately modify the deposit and both sets of tokens "inherited" from the former step.

Nie wiem dlaczego, ale nie do końca jestem pewien definicji create-twin. Z drugiej jednak strony wydaje się dość oczywiste, że taka funkcja musi dać się dobrze zdefiniować. ???

Now we are ready to define the function of passing actual to formal parameters. Let cl-ide be the name of a class where our procedure is being declared:

```
pass-actual : ForParDen x ForParDen x ActParDen x ActParDen x Identifier ⟼
                                                        ⟼ Env ⟼ Store ⟼ Store
pass-actual.(fpd-v, fpd-r, apd-v, apd-r, cl-ide).dt-env.ct-sto =
   is-error.ct-sto          ➔ ct-sto                                   call time store
1. checking the static compatibility of parameters
   let
      message  = statically-compatible.(fpd-v, fpd-r, apd-v, apd-r)
   message ≠ 'OK'           ➔ ct-sto ◄ message
2. identifying the identifiers, values, and references of actual and formal parameters
   let
      for k, n ≥ 0
      ((ide-fv.i, ted-fv.i) | i=1;k)        = list-of-for-par.fpd-v          (see Sec. 6.6.3.3)
      ((ide-fr.i, ted-fr.i) | i=1;n)        = list-of-for-par.fpd-r
      (ide-av.i | i=1;k)                    = apd-v
      (ide-ar.i | i=1;n)                    = apd-r
      (ct-obn, ct-dep, ct-ota, ct-sft, 'OK'))  = ct-sto                        call-time store
      (dt-cle, dt-pre, dt-cov)              = dt-env            declaration-time environment
```

ct-obn.ide-av.i = ? ➔ ct-sto ◄ 'actual val. parameter not declared' for i = 1;k
ct-obn.ide-ar.i = ? ➔ ct-sto ◄ 'actual ref. parameter not declared' for i = 1;n
let
ref-av.i = ct-obn.ide-av.i for i=1;k the references of actual value-parameters
ref-ar.i = ct-obn.ide-ar.i for i=1;n the references of actual reference-parameters
ct-dep.ref-av.i = ? ➔ ct-sto ◄ 'actual val. parameter not initialized'[49] for i = 1;k
let
val-av.i = ct-dep.ref-av.i for i = 1;k
(dat-av.i, typ-av.i) = val-av.i for i = 1;k
(tok-av.i, (typ-rav.i, yok-rav.i, ota-av.i)) = ref-av.i for i = 1;k
(tok-ar.i, (typ-rar.i, yok-rar.i, ota-ar.i)) = ref-ar.i for i = 1;n

3. *computing the types of formal parameters*

let
de-typ-fv.i = ted-fv.i.(dt-env, ct-sto) declared types of formal value-parameters for i = 1;k
de-typ-fr.i = ted-fr.i.(dt-env, ct-sto) declared types of formal reference-parameters for i = 1;n
de-typ-fv.i : Error ➔ ct-sto ◄ d-typ-fv.i for i = 1;k
de-typ-fr.i : Error ➔ ct-sto ◄ de-typ-fr.i for i = 1;n

4. *creating twins of formal value-parameters*

let
sft-0 = ct-sft
snt-0 = { }
dep-0 = ct-dep
(twin-val-av.1, dep-1, snt-1, sft-1) = create-twin.(val-av.i, dep-0, snt-0, sft-0)
. . .
(twin-val-av.k, dep-k, snt-k, sft-k) = create-twin.(val-av.i, dep-(k-1), snt-(k-1), sft-(k-1))

5. *creating references for formal value-parameters*

let
((tok-fv.1,…,tok-fv.k), snt, new-sft) = get-new-tok.(k, { }, sft-k)[50]
new-ref-fv.i = (tok-fv.i, (de-typ-fv.i, yok-av.i, ota-av.i)) for i = 1;k

6. *checking type acceptance of:*

actual value-parameters by formal value-parameters
not de-typ-fv.i **TTA.dt-cov** typ-av.i ➔ sta ◄ 'value parameters not compatible' for i = 1;k
actual reference-parameters by formal reference-parameters
not de-typ-fr.i **TTA.dt-cov** typ-rar.i ➔ sta ◄ 'reference parameters not compatible' for i = 1;n

7. *creating a local initial objecton of the store*

let
li-obn-fv = [ide-fv.1/new-ref-fv.1,…,ide-fv.k/new-ref-fv.k] the binding of formal val-param.
li-obn-fr = [ide-fr.1/ref-ar.1,…,ide-fr.n/ref-ar.n] the binding of formal reference-parameters
li-obn = li-obn-fv ♦ li-obn-fr local initial objecton

8. *creating a local initial object deposit*

li-dep = dep-k[new-ref-fv.1/twin-val-av.1,…,new-ref-fv.k/twin-val-av.k] bind. twins to new ref.

9. *creating a local initial store*

li-sto = (li-obn, li-dep, cl-ide, new-sft, 'OK')

10. *creating a local initial state*

li-sta = (dt-env, li-sto)
true ➔ li-sta

---

[49] Mathematically we could have assumed that not initialized value parameters are allowed, but such parameters would have not too much of a practical sense, since they would act as local variables. If, therefore, a not initialized parameter is passed to a procedure call, we may have a justified supposition that it is a programmer's mistake, rather than an intention. Consequently we signalize an error. The situation with reference parameters is, of course, different.

[50] In this place we use the function **get-new-tok** to generate tokens for the future reference of formal value parameters. In this context we do not need to monitor a set of new tokens but our function must be given such a set as an argument. Since it can be an arbitrary set, we choose the empty set { } to play this role.

Function pass-actual performs the following steps:

1. checks the static compatibility of actual with formal parameters,
2. identifies/computes:
   a. the identifiers of actual and formal parameters,
   b. the references of actual parameters; if they are not declared, then an error is signalized,
   c. the values of actual-value parameters; if they are not initialized, then an error is signalized; note that actual reference parameters need not be initialized,
3. computes the types of formal parameters; these types are computed in a state carrying a declaration-time environment, but the types declared in this environment (in its classes) are the same as the types in the call-time environment[51],
4. creates twin values for formal value-parameters; for not-object values twins are just these values, whereas the twins of object differ from the source objects only in internal tokens (rule 7. in Sec. 5.4.3),
5. creates new references for formal value-parameters; they get fresh tokens, declared types of formal parameters, and the yokes and origins of actual parameters,
6. checks the acceptance for declaration-time covering relation of:
   a. types typ-av.i of the values of actual value-parameters by the declared types de-typ-fv.i of corresponding formal parameters,
   b. types typ-rar.i of the references of actual reference-parameters by the declared types de-typ-fr.i of corresponding formal parameters,
7. creates a local initial objecton by assigning created references to formal value-parameters, and the (old) references of actual reference-parameters to formal reference-parameters; the actual values are accepted by the new references, because they differ from the "old" references by tokens only,
8. creates a local initial deposit as an extension of the call-time deposit by new references bound to the values of actual value-parameters; note that all "old" references of actual value parameters remain bounded in the new deposit, but now they are orphan references, which means that we can't access them from the local state,
9. creates a local initial store with new objecton, new deposit, new set of free tokens, and cl-ide (the name of the hosting class) as its origin tag (rule 6 in Sec. 5.4.3),
10. creates a local initial state by putting together the declaration-time environment with the local initial store.

Two remarks are necessary about the required compatibility between the profiles of actual and formal parameters:

- In the case of value parameters, the references of formal parameters ref-fv's are created by the passing parameters mechanism (PPM) in such a way that the declared types of parameters become the types of these references. Then PPM assigns to these references the values of actual parameters, and therefore the types of these values — which may be different from the declared types — must be acceptable by the latter.
- In the case of reference parameters the situation is different. Now, PPM assigns the references of actual parameters ref-ar's to formal parameters ide-fr's. Then we only request that the types of ref-ar's are accepted by the declared types. However, since **TTA.cov** is, by definition, transitive, also now declared types will accept the types of actual values.

Notice that the described mechanism of creating local initial stores does not offer a possibility of using global variables/attributes, i.e. variables/attributes visible both outside and inside procedure-bodies. All "external interventions" of a procedure call must be realized by reference parameters, and therefore, must be explicitly declared. In our opinion such a solution contributes to the clarity of programs, and also simplifies construction rules of correct programs with procedure calls (cf. Sec. 9.4.6.3).

6.6.3.5 Returning the references of reference parameters

By the end of the execution of a procedure call we reach a *local terminal state* that consists of:

---

[51] It is the consequence of our assumption of Sec. 6.3 that all class declarations precede in programs all instructions.

- a *local terminal store*, where the objecton binds formal parameters and possibly some local variables declared in the body of the procedure,
- a *local terminal environment*, where possibly new local classes, pre-procedures and procedures have been declared, and/or the covering relation has been augmented by new pairs of types.

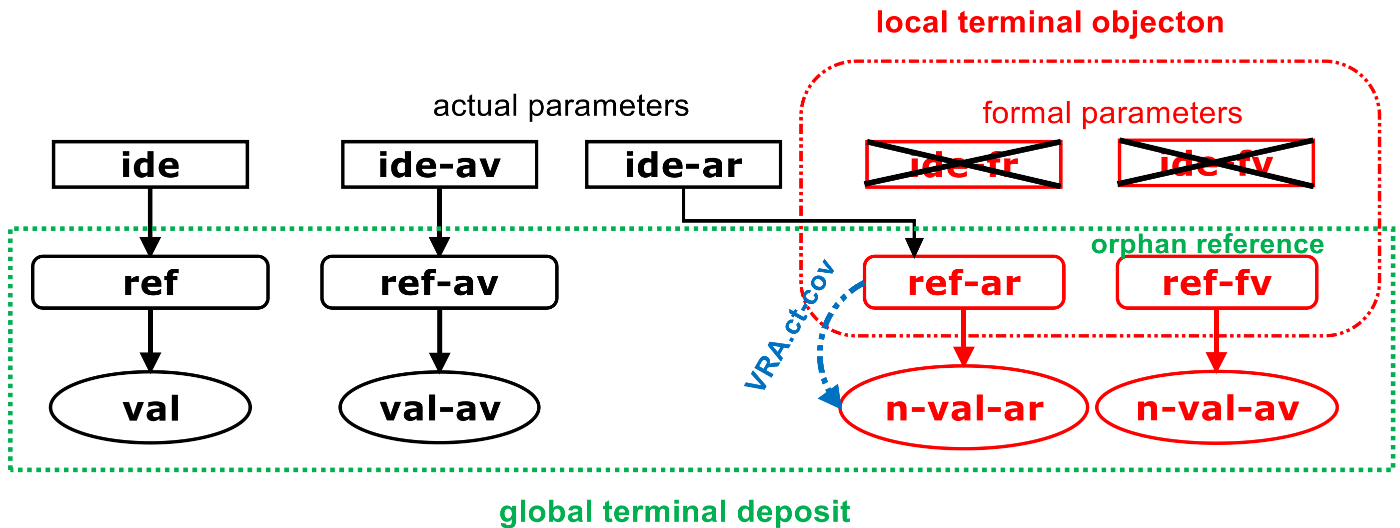


Fig. 6.6-3 Returning references to actual reference-parameters of a procedure

Next, the exiting mechanism builds a *global terminal state* (Fig. 6.6-1) consisting of a call-time environment — all locally declared classes and procedures, and new pairs of types in covering relation cease to exist — and a *global terminal store* where actual reference parameters regain their call-time references. This store consists of (Fig. 6.6-3):

- *global call-time objecton* — where all global variables, including actual parameters, "regain visibility", and all formal parameters and local variables cease to exist; actual reference-parameters point (back) to their call-time references,
- *local terminal deposit* — this deposit is passed unchanged to the global store, but the (created by the call) references of formal value-parameters and of local variables become orphan references; for the sake of simplicity we do not introduce a garbage-collection mechanism,
- *global call-time origin tag of the store* — note that the call-time origin tag may be: (1) either public-visibility origin tag $, or (2) a class name; case (2) will happen, if our procedure was locally declared and called in the body of another procedure,
- *local terminal set of used references* — since we do not introduce a garbage collection mechanism, no formerly used tokes are released (simplification of the model).

Before we formalize the mechanism of returning reference parameters, we introduce an auxiliary (meta) predicate to be used at the exit of the procedure call in checking, if in the local terminal state:

A. all formal reference parameters are initialized (explanation below),
B. their values are acceptable by their references in the context of the call-time covering relation.

Of course, B. is a necessary condition for the global terminal store to be well-formed. As was already mentioned, condition B. may be unsatisfied, if the following situation takes place:

a. ide → (tok, (r-typ, yok, ota)) → (dat, v-typ) and

b. (r-typ, v-typ) : lt-cov but
c. (r-typ, v-typ) /: ct-cov

Note that case c. may happen if the local covering relation has been (locally) enriched by the pair (r-typ, v-typ).

Parameters (identifiers) which satisfy A. and B. will be called *adequate* in a given state. This property is formalized by the following predicate-like function:

```
adequate : Identifier ⟼ WfState ⟼ {tt, ff} | Error
adequate.ide.sta =
   is-error.sta              ➔ error.sta
   let
      ((cle, pre, cov), (obn, dep, ota, sft, 'OK')) = sta
   obn.ide          = ?   ➔ 'parameter not declared'
   dep.(obn.ide)    = ?   ➔ 'parameter not initialized'
   let
      ref = obn.ide
      val = dep.ref
   ref VRA.cov val           ➔ tt
   true                      ➔ ff
```

Having this function we can describe the mechanism of returning formal parameters:

```
return-formal : ForParDen ⟼ Store ⟼ Store ⟼ Env ⟼ Store
return-formal.fpd-r.ct-sto.lt-sto.dt-env =
   is-error.lt-sto                                  ➔ lt-sto
   let
      (ct-obn, ct-dep, ct-ota, ct-sft, 'OK') = ct-sto
      (lt-obn, lt-dep, lt-ota, lt-sft, 'OK')   = lt-sto
      (ide-fr.i | i=1;n)                       = list-of-ide.(list-of-for-par.fpd-r)
   adequate.(ide-fr.i).(dt-env, lt-sto) : Error ➔ ct-sto◄ adequate.(ide-fr.i).(dt-env, lt-sto) for i = 1;n
   adequate.(ide-fr.i).(dt-env, lt-sto) = ff    ➔ ct-sto◄ 'ide-fr.i not adequate'        for i = 1;n
   true                                         ➔ (ct-obn, lt-dep, lt-sft, ct-ota, 'OK')   for i = 1;n
```

The output store is a combination of:

- call-time objecton, and call-time origin tag,
- local-terminal deposit, and local-terminal set of free tokens.

Two facts are to be emphasized:

1. We do not allow a returned reference parameter to be not initialized condition A.). Note that this could have happened since we allow passing noninitialized actual reference parameters to formal parameters (cf. Sec. 6.6.3.4). However, whereas not initialized reference parameters at the entrance of a procedure call make sense, if the same happens at the exit, such parameters turn out to be useless. Since it is rational to expect that a programmer may introduce useless parameters only by mistake, we make an engineering decision to signalize an error whenever such a situation happens.
2. It could have happened that the value of a formal reference parameter ide-fr.i is of a type which was accepted by its reference for local covering relation, but is not accepted for global (call-time) relation. In such a case an error message should be generated. To do so, we use function adequate.

In point 2. we may see a problem, since formally this function gets a declaration-time environment (cf. Sec. 6.6.3.2), hence also a declaration-time covering relation dt-cov, and what we intend to do, is to check the adequacy for call-time covering relation ct-cov. Note, however, that by assumption made in Sec. 6.3, if a procedure is called prior to open procedures, then its call will abort, since our procedure is "not yet declared", and therefore we do not need "to care" about covering relation. On the other hand, if procedure is

called after open procedures, then we have the equality ct-cov = dt-cov, since no declaration may appear after open procedure.

6.6.3.6 Calling an imperative procedure

The calls of imperative procedures are atomic instructions. Their mechanism is described by the following constructor:

```
call-imp-pro : Identifier x Identifier x ActParDen x ActParDen ⟼ InsDen
call-imp-pro : Identifier x Identifier x ActParDen x ActParDen ⟼ WfState ⟶ WfState
call-imp-pro.(cl-ide, pr-ide, apd-v, apd-r).ct-sta =
   is-error.ct-sta                     ➔ ct-sta
   let
      (ct-env, ct-sto)  = ct-sta                                      call-time state
      (ct-cle, ct-pre)  = ct-env
   ct-pre.(cl-ide, pr-ide) = ?         ➔ ct-sta ◄ 'procedure-unknown'
   ct-pre.(cl-ide, pr-ide) /: ImpPro   ➔ ct-sta ◄ 'imperative-procedure-expected'
   let
      ipr = ct-pre.(cl-ide, pr-ide)
   ipr.(apd-v, apd-r).ct-sto = ?       ➔ ?
   let
      gt-sto = ipr.(apd-v, apd-r).ct-sto                        global terminal store
   true                                ➔ (ct-env, gt-sto)
```

To call a procedure, we first seek it in the procedure environment of a state (cf. Sec. 6.6.1), and then we apply it to the current (i.e. call-time) store. We recall that the terminal store may be the call-time store with an error (cf. Sec. 6.6.3.2).

6.6.4 Functional pre-procedures

6.6.4.1 Creating functional pre-procedures

Similarly to imperative pre-procedures, and for the same reason, we build *functional pre-procedures*. This process is formalized in the following definition:

```
create-fun-pre-pro : FunProSigDen x ProDen x ValExpDen x Identifier ⟼ FunPrePro
create-fun-pre-pro : FunProSigDen x ProDen x ValExpDen x Identifier ⟼
                                       ⟼ Env ⟼ ActParDen ⟼ Store → ValueE
create-fun-pre-pro.(fps,  prd, ved, cl-ide).dt-env.apd.ct-sto =    dt- creation time, ct- call time
   is-error.ct-sto                ➔ error.ct-sto
   let
      (ct-obn, ct-dep, ct-sft, ct-ota, 'OK') = ct-sto
      (fpd, ted)                             = fps           functional-procedure signature
      li-sto = pass-actual.(fpd, (), apd, (), cl-ide ).dt-env.ct-sto          local initial store
   is-error.li-sto                ➔ error.li-sto
   let
      li-sta    = (dt-env, li-sto)                                          local initial state
      ex-typ    = ted.li-sta                          the expected type of the returned value
   ex-typ : Error                 ➔ ex-typ
   (prd • ved).li-sta= ?          ➔ ?
   (prd • ved).li-sta : Error     ➔ (prd • ved).li-sta
   let
      (cor, typ) = (prd • ved).sta-li                                      lt- local terminal
   not ex-typ TTA.ct-cov typ ➔ 'types-incompatible'
   true                           ➔ (cor, ex-typ)
```

This constructor is defined analogously to the corresponding constructor of imperative pre-procedures. The execution of its body consist in the evaluation of the argument value-expression ved in an output state of the argument program prd. In this way we get an intermediate value (cor, typ), but the value finally returned by the procedure is (cor, ex-typ), where ex-typ is the type expected by the procedure.

Note that we can't issue simply (cor, typ), because, if typ has been locally declared, it will not be seen in the global environment. Of course, before we output (cor, ex-typ) we have to check if its type accepts typ.

6.6.4.2 Calling functional procedures

The constructor corresponding to the calls of deep functional procedures is the following:

```
call-fun-pro : Identifier x Identifier x ActParDen  ↦ValExpDen
call-fun-pro : Identifier x Identifier x ActParDen  ↦ WfState ⟶ Value | Error
call-fun-pro.(cl-ide, pr-ide, apd).ct-sta =                              ct- call time
   is-error.ct-sta                    ➔ ct-sta
   let
      ((cle, pre, cov), ct-sto) = ct-sta
   pre.(cl-ide, pr-ide) = ?           ➔ 'procedure-unknown'
   pre.(cl-ide, pr-ide) /: FunPro     ➔ 'functional-procedure-expected'
   let
      fpr = pre.(cl-ide, pr-ide)
   fpr.apd.ct-sto = ?                 ➔ ?
   true                               ➔ fpr.apd.ct-sto
```

The called procedure is selected from the procedure environment, and then it is applied to the (call-time) store of the current state. If this application terminates successfully, then the outputted value becomes the output of the call. Note that fpr.apd.ct-sto may be an error.

6.6.5 Object pre-constructors

6.6.5.1 Object constructors versus imperative procedures

Similarly as in many OO languages, objects are created in our model exclusively by dedicated imperative procedures called *object constructors*. For a class MyClass named 'MyClass' an object constructor associated with this class (declared in this class) is a function that given a list of actual parameters, and the name ide of the future object, returns a store-to-store function that performs three major steps:

1. it creates an object of a class MyClass whose objecton is a twin of the objecton of MyClass, and whose type is 'MyClass',
2. it (optionally) modifies the current deposit, by changing the values assigned to the attributes of the new object; to do this it uses a program,
3. it assigns new object to the reference of ide in the deposit of the current store; the objecton of the new object is a sibling of the objecton of MyClass.

Since we do not want object constructors to have side effects (an engineering decision), we assume that they get only value parameters.

oco : ObjCon = ActParDen x Identifier ↦ Store → Store

Since object constructors are regarded as procedures, by an analogy to pre-procedures, we introduce *object pre-constructors* with the following domain:

opc : ObjPreCon = Env ↦ ObjCon

As we see, although the calls of object constructors are instructions, like the calls of imperative procedures, object constructors themselves are different from imperative procedures.

6.6.5.2 Creating an object pre-constructor

The creator of object pre-constructors given formal parameter denotations, a name of a class and a program denotation, returns an object pre-constructor. The latter given an environment returns an object constructor, that given the denotations of actual value parameters, an a name of the future object, and a (call time) store, returns a new store where the object name points to a new object of the type of the given class. The objecton of the created object is a sibling of the objecton of the involved class.

```
create-obj-pre-con : ObjConSigDen x ProDen ⟼ ObjPreCon
create-obj-pre-con : ForParDen x Identifier x ProDen ⟼
                                    ⟼ Env ⟼ ActParDen x Identifier ⟼  Store → Store
create-obj-pre-con.((fpd, cl-ide), prd).dt-env.(apd, ob-ide,).ct-sto =      cl-ide class identifier
  is-error.ct-sto                    ➔ ct-sto                              ob-ide object identifier
```

1. *parent class is identified*

```
  let
    (dt-cle, dt-pre, dt-cov)              = dt-env                 declaration-time environment
    (ct-obn, ct-dep, ct-ota, ct-sft, 'OK') = ct-sto                           call-time store
  dt-cle.cl-ide        = ?           ➔ ct-sto ◄ 'parent class not declared'
  ct-obn.ob-ide        = ?           ➔ ct-sto ◄ 'object-identifier must be declared'
  ct-dep.(ct-obn.ob-ide) = !         ➔ ct-sto ◄ 'object-identifier must not be initialized'
```

2. *future reference of the constructed object is identified*

```
  let
    (tok, (typ, yok, ob-ota))  = ct-obn.ob-ide        future reference of the constructed object
    (ide, tye, mee, cl-obn)    = dt-cle.cl-ide               parent class of the future object
  not typ TTA.ct-cov cl-ide          ➔ 'types not compatible'
```

3. *formal-parameter store is created*

```
  let
    fp-sto = pass-actual.(fpd, (), apd, (), cl-ide).dt-env.ct-sto          formal-parameter store
  is-error.fp-sto                    ➔ ct-sto ◄ error.fp-sto
  let
    (fp-obn, fp-dep, cl-ide, fp-sft, 'OK') = fp-sto
  dom.cl-obn ∩ dom.fp-obn ≠ {}       ➔ ct-sto ◄ 'a clash between parameters and attributes'
```

4. *local initial state is created*

```
  let
    (tw-obn, li-sft)    = create-twin.(cl-obn, fp-sft)                  tw-obn twin objecton
    li-obn              = fp-obn ♦ tw-obn
    li-sto              = (li-obn, fp-dep, cl-ide, li-sft, 'OK')
    li-sta              = (dt-env, li-sto)                                  local-initial state
  prd.li-sta = ?                     ➔ ?
```

5. *local initial state is transformed into a local terminal state*

```
  let
    lt-sta = prd.li-sta                                                  local terminal state
  is-error.lt-sta                    ➔ ct-sto ◄ error.lt-sta
```

6. *resulting object and terminal global store are created*

```
  let
    (lt-env, (lt-obn, lt-dep, lt-ota, lt-sft, 'OK')) = lt-sta
    re-obn  = truncate.(lt-obn, dom.cl-obn)          resulting objecton
    re-obj  = (re-obn, cl-ide)                       resulting object
    ob-ref  = ct-obn.ob-ide                          future reference of the resulting object
```

```
      tg-dep  = ct-dep[ob-ref/re-obj]                  terminal global deposit
   true                                  ➔ (ct-obn, tg-dep, ct-ota, lt-sft, 'OK')
```

Similarly to the former creators also this one builds a pre-constructor, that given a declaration-time environment returns a class constructor. The latter, given a call-time store returns this store with a resulting object bound in it. The resulting object is derived from the objecton of an indicated class, which is built into the pre-constructor. This action is performed in the following ten steps (we skip commenting errors):

1. A parent class indicated by its name cl-ide is identified.
2. The future reference of the new object is identified. It is the reference of the predeclared object variable ob-ide. The type of this reference must be compatible with the type of the created object, i.e. with cl-ide.
3. We create a formal-parameter store fp-sto, where only formal value parameters are bound in the objecton. The reference parameters are not involve, and the created store binds only formal value parameters.
4. The creation of a local initial state:
   4.1. We create a twin objecton tw-obn of the objecton cl-obn of the parent class.
   4.2. We create a local initial objecton li-obn by combining (overwriting) the twin objecton with the objecton of the formal-parameter store. The only attributes of this objecton are formal parameters and the attributes of the twin objecton. Note that these two sets of identifiers must be disjoint.
   4.3. We create a local initial store by combining the declaration-time environment with the local initial store. Note that the origin tag of this store is cl-ide.
   4.4. Local initial state li-sta is composed of the declaration-time environment, and the local initial store
5. The local initial state is transformed by the program prd to a local-terminal state lt-sta.
6. The creation of the resulting object and terminal global store
   6.1. The local terminal objecton is truncated to the attributes of the objecton of the class cl-obn thus giving a resulting objecton re-obn. This objecton is then used to create the resulting object re-obj by adding to it its type cl-ide.
   6.2. A terminal global deposit of the store is created by binding the created object to its reference ob-ref. Note that the compatibility of the types of ob-ide and re-obj has been checked in point 2.
   6.3. A terminal global deposit tg-dep is created by assigning the created object to ob-ide in the call-time deposit.
   6.4. The terminal global store includes the call-time objecton, the new deposit, the call-time covering relation (an engineering decision), a call-time origin tag, and a local terminal set of free tokens.

Returning to our example of an object constructor ConstructObject of Sec. 5.1, the program involved in the creation of a new object by this constructor consists of a pair of assignment statements:

```
no    := number + 1;
next  := node
```

where no and next are object's attributes, and number an node are formal parameters.

6.6.5.3 Calling object constructors

Calls of object constructors are instructions that create new objects and assign them to predeclared variables. The constructor of such calls takes a name of a class cl-ide where the a constructor has been declared, the name va-ide of this constructor, an identifier ob-ide to which the new object will be assigned, a list of actual value parameters, and returns an instruction denotation:

```
call-obj-con : Identifier x Identifier x Identifier x ActParDen ⟼ InsDen
call-obj-con : Identifier x Identifier x Identifier x ActParDen ⟼ WfState ⟶ WfState
call-obj-con.(ob-ide, cl-ide, va-ide, apd-v).sta =
   is-error.ct-sta                         ➔ ct-ct-sta
   let
```

```
    ((ct-cle, ct-pre), ct-sto) = ct-sta                                    call-time state
  ct-pre.(cl-ide, va-ide) = ?             ➔ ct-sta ◄ 'constructor unknown'
  ct-pre.(cl-ide, va-ide) /: ObjCon       ➔ ct-sta ◄ 'object constructor expected'
  let
    oco = ct-pre.(cl-ide, va-ide)
  oco.(apd-v, ob-ide).ct-sto = ?          ➔ ?
  let
    new-sto = oco.(apd-v, ob-ide).ct-sto
  true                                    ➔ (ct-env, new-sto)
```

The instructions of object-constructor-calls may be said to be "hybrid instruction", since they are half instructions and half declarations. They are half declarations because they declare new object variables. However, structurally they have been included into the domain of instruction, to allow them to be iterated.

6.7 Declarations

6.7.1 An overview of declarations

Declarations in our model may act at two different levels:

1. at the **level of states**
   a. they assign references, classes and procedures to identifiers,
   b. they modify covering relations,
2. at the **level of classes**
   a. they assign references, pre-procedures and types to identifiers,
   b. they assign values to references in deposits.

The assignment of a reference to an identifier is usually referred to as a *variable* or an *attribute declaration*.

The declarations of classes will be executed in two steps:

- the choice of a parent class which may be either empty or previously declared,
- the removal from this class of all object pre-constructors (an engineering decision[52]) and an enrichment of the resulting class by new items with the help of class transformers.

At the stage of class transformations we may:

- add abstract items,
- concretize abstract items, i.e. replace abstract items by corresponding concrete ones,
- add concrete items.

When we create a new class in a state, we modify the class environment of this state, and additionally, if we declare a concrete attribute in this class, or if we concretize an abstract attribute, then we modify also the store of the state by modifying its deposit.

Class transformations are performed by class transformers (Sec. 6.7.4) that modify classes stored in class environments of states. However, the fact that we have class transformers in our language does not mean that we can modify declared classes. As we are going to see, class transformers will be used exclusively within class declarations, which means that classes once declared, will never be changed.

As we have assumed in Sec. 6.3, all declaration in our programs will syntactically precede all instructions. This rule concerns in particular covering relations, and, in fact, this is why we include them in the category of declarations rather than instructions. The second reason of this decision is that the modifications of covering

---

[52] It is usual in the existing programming languages that object constructors are not inherited by children classes from their parent classes. This rule seems rather obvious — when we create an object of a class, we use an object constructor of that class.

relations will be restricted to their enrichments by new pairs of types. All these decisions do not affect the functionality of our programs, but significantly simplify the rules of building correct programs (Sec. 9.4). In this way we realize the Second Principle of Simplicity formulated in Sec. 3.3.

Now, consider the following example of a class declaration written in an anticipated concrete syntax of our language (Sec. 7.3):

```
class MyClass:
   par HisClass by:
      let age be integer and private tel;
      set worker as HisClass.employee and private tes ;
      proc promote(val cfp ref cfp) prc corp
ssalc
```

This declaration enriches incrementally a previously declared parent class HisClass, by one private integer attribute age, one type constant worker, whose declaration refers to a class constant of the parent class, and one imperative procedure (concrete method).

Since the declarations of types and pre-procedures will be hidden in class transformers, formally we are going to have five categories of atomic declarations with the following constructors:

var-dec	: ListOfIde x TypExpDen	⟼ DecDen	variable declarations
enrich-cov-rel	: TypExpDen x TypExpDen	⟼ DecDen	enrichments of cov. rel.
cla-dec	: Identifier x ClaInd x ClaTraDen	⟼ DecDen	class declarations
pro-open	:	⟼ DecDen	procedure opening
skip-dec	:	⟼ DecDen	trivial declaration

Since declarations can be composed sequentially, we introduce also a corresponding constructor:

compose-dec : DecDen x DecDen ⟼ DecDen

We omit obvious definition of the last constructor.

Once all classes have been declared in a program, we perform a one-step operation called the *opening of procedures* (Sec. 6.7.6) that creates procedures, functions and object constructors out of the corresponding pre-procedures, pre-functions and object pre-constructors respectively, and then assigns them to their names in the procedure environment of the current state.

6.7.2 Declarations of variables

Our variable declarations declare list of variables of a common type and yoke. For the sake of simplicity we assume that variable declarations do not initialize variables. They only assign references to identifiers. The initialization of variables must be done by assignment instructions (Sec. 6.4). The definition of our constructor is following (we recall that yoke-expression denotations are just yokes; cf. Sec. 6.4.2):

```
var-dec : ListOfIde x TypExpDen x YokExpDen ⟼ DecDen                i.e.
var-dec : ListOfIde x TypExpDen x YokExpDen ⟼ WfState ⟼ WfState
var-dec.(loi, ted, yok).sta =
   is-error.sta                          ➔ sta
   loi = ()                              ➔ sta ◄ 'empty list of variables can't be declared'
   let
      ((cle, pre, cov), (obn, dep, ota, sft, 'OK'))   = sta
      (ide-1,…,ide-n)                                 = loi
   ted.sta : Error                       ➔ sta ◄ ted.sta
   declared.ide-i.sta                    ➔ sta ◄ 'identifier declared'      for i = 1;n
   let
      de-typ                  = ted.sta                                     declared
      type
      (tok-1, sft-1)          = get-tok.sft
```

(tok-i, sft-i)) = get-tok.sft(i-1) for i = 2;n
ref-i = (tok-i, (de-typ, yok, $))
de-typ : ObjTyp **and** cle.de-typ = ? ➔ 'class unknown'
true ➔ ((cle, pre, cov), (obn[ide-i/ref-i | i = 1;n], dep, ota, sft-n, 'OK'))

Note that if we declare an object variable, we check if the corresponding type is a declared name of a class. This rule is in contrast to the case where we add an abstract object-attribute to a class. In that case, as we are going to see in Sec. 6.7.4.2, we shall allow that the type of an attribute, i.e. the name of a corresponding class, may be not (yet) declared. Such a solution is necessary to allow anticipatory referencing in classes.

Another fact to be noted is that variables are always public (an engineering decision), which means that their origin tags are equal to $.

6.7.3 Declarations of classes — a basic constructor

As has been already said, classes are declared in three steps:

1. In the first step we identify an initial *parent class* which is either an empty class or an earlier declared one.
2. In the second step we generate a *funding class*. If parent class was empty then the funding class is just this empty class. Otherwise we take a declared earlier parent class and remove from it all types, pre-procedures and object pre-constructors (an engineering decision). We also replace its name by a new one. What is inherited by funding class from parent class are, therefore, its attribute and signatures[53]. Note, however, that the inheritance of attributes means also the inheritance of their references, and therefore also values assigned to them in the deposit. On the other hand, as we are going to see later, the initial values of attributes may be changed at later stages of the declaration execution.
3. In the third step, we apply a class transformer (Sec. 6.7.4) to enrich funding class by new attributes, types, procedure signatures and pre-procedures.

We recall (Sec. 6.1) that the domain of class-transformer denotations is the following:

ctc : ClaTraDen = Identifier ⟼ WfState → WfState.

Here some methodological remark are in order to explain why class transformers modify states rather than classes, and why they take an identifier as an argument?

In an earlier version of our model class transformers were functions transforming classes and states, rather than classes "in states":

ClaTraDen = (Class x WfState) → (Class x WfState)

The modification of states by class transformers is necessary to describe the initialization of class attributes, whose values are created by the evaluation of value expressions. From a denotational perspective, this model worked quite well. Still, as turned out later, it led to technical problems in the definitions of rules for the creation of correct class declarations (Sec. 9.4.4.2). This example shows that in designing a programming language, we should consider not only the simplicity of its denotational model but also an ease of building program-construction rules (cf. Sec. 3.3).

Once we have assumed that class transformers will not get "input classes" as their arguments, we had to indicate these classes in a different way. Class identifiers were an obvious choice to play this role.

In the end, to define our constructor of class-declaration denotations we shall need an auxiliary function to create funding classes from parent classes:

make-funding-class : ClaInd ⟼ Identifier ⟼ WfState ⟼ Class | Error

---

[53] In typical programming languages (??? Janusza prosimy o przykłady) funding class inherits from parent class all items except object pre-constructors. However, since in our model we have assumed (cf. Sec. 5.1) that types and procedures are public, we may access them independently of the class where they have been declared. Making their copies under different would not have much o a practical sense.

```
make-funding-class.cli.ide.sta =
   cli = 'empty-class'  ➔  (ide, [ ], [ ], [ ])
   let
      ((cle, pre, cov), sto) = sta
   cle.cli = ?              ➔ 'parent class unknown'
   let
      (cl-ide, tye, mee, obn)                                  = cla.cli        parent class
      [ide-1/ppr-1,…,ide-n/ppr-n, ide-(n+1)/sig-1,…,ide-(n+k)/sig-k]  = mee
   true ➔                 (ide, [ ], [ide-(n+1)/sig-1,…,ide-(n+k)/sig-k], obn
```

If the class indicator is 'empty-class', then the funding class is an empty class with ide as its internal name. Otherwise the funding class in the class indicated by cli and modified by giving it ide as a new name, removing all types from its type environment, and removing all pre-procedures from its method environment. The constructor of class-declaration denotations is now the following:

```
cla-dec : Identifier x ClaInd x ClaTraDen ↦ DecDen
cla-dec : Identifier x ClaInd x ClaTraDen ↦ WfState ↦ WfState
cla-dec .(de-ide, pa-cli, ctc).sta =                 de- for "declared"; pa- for "parent"
   is-error.sta           ➔ sta
   declared.de-ide.sta    ➔ sta ◄ 'identifier already declared'
   let
      ((cle, pre, cov), sto)   = sta
      fu-cla                   = make-funding-class.pa-cli.de-ide.sta        funding class
   fu-cla : Error         ➔ sta ◄ fu-cla
   let
      fu-sta = ((cle[de-ide/fu-cla], pre, cov), sto)                         funding state
   ctc.de-ide.fu-sta = ?  ➔ ?
   let
      res-sta  = ctc.de-ide.fu-sta          resulting state (with an enriched funding class)
   is-error.res-sta       ➔ sta ◄ error.res-sta
   true                   ➔ res-sta
```

The declaration of a class starts from making a funding class, and assigning it de-ide to class environment in an intermediate state called a *funding state*. This state, hence the funding class in this state, is then modified by the argument class transformer ctc. Note that the first argument of ctc is de-cla, i.e., the name of the modified class.

6.7.4 Class transformers

6.7.4.1 The signatures of constructors

Classes are transformed by adding to them new attributes, types or methods. Technically we bind new identifiers in class objectons, in type environments or in method environments. In all cases we have three options: we can add an abstract item, a concrete item, or we can concretize an abstract item. The last option may be used when we build a child of an earlier declared class. Finally, class transformers may be composed sequentially.

The domain of class-transformer denotations has been defined in Sec. 6.7.3. The list of constructors of class-transformer denotations, is the following, where abs means "abstract" and con means "concrete".

```
add-abs-att          : Identifier x TypExpDen x PriSta                 ↦ ClaTraDen
concretize-abs-att   : Identifier x ValExpDen                          ↦ ClaTraDen
add-con-att          : Identifier x ValExpDen x TypExpDen x PriSta     ↦ ClaTraDen

add-abs-typ          : Identifier                                      ↦ ClaTraDen
concretize-typ       : Identifier x TypExpDen                          ↦ ClaTraDen
add-con-typ          : Identifier x TypExpDen                          ↦ ClaTraDen
```

```
add-abs-imp-met       : Identifier x ImpProSigDen                          ⟼ ClaTraDen
concretize-imp-met    : Identifier x ImpProSigDen x ProDen                 ⟼ ClaTraDen
add-con-imp-met       : Identifier x ImpProSigDen x ProDen                 ⟼ ClaTraDen

add-abs-fun-met       : Identifier x FunProSigDen                          ⟼ ClaTraDen
concretize-fun-met    : Identifier x FunProSigDen x ProDen x ValExpDen     ⟼ ClaTraDen
add-con-fun-met       : Identifier x FunProSigDen x ProDen x ValExpDen     ⟼ ClaTraDen

add-abs-obj-met       : Identifier x ObjConSigDen                          ⟼ ClaTraDen
concretize-obj-met    : Identifier x ObjConSigDen x ProDen                 ⟼ ClaTraDen
add-con-obj-met       : Identifier x ObjConSigDen x ProDen                 ⟼ ClaTraDen

compose-cla-tra       : ClaTraDen x ClaTraDen                              ⟼ ClaTraDen
```

In the following sections, we will show some examples of the definitions of these constructors. Each of these constructors will perform three similar steps:

1. it will identify a class assigned to cl-ide,
2. it will appropriately modify this class,
3. it will assign the new class to cl-ide in the class environment of the current state.

Transformers built by the first fifteen constructors will be called *atomic transformers*, to contrast them from *composed transformers* built by means of compose-cla-tra.

6.7.4.2 Adding an abstract attribute to the objecton of a class

Contrary to variables that are, as a rule, private (Sec. 6.7.2), when we declare a class attribute we have to decide about its visibility status. To incorporate the privacy mechanism into the declarations of class attributes, we introduce a new domain, and an auxiliary function. The new domain includes two marks defining privacy status

pst : PriSta = {‘private’, ‘public’}

Adding an abstract attribute to a class consists in adding a new attribute to class objecton. The corresponding constructor is similar to the declaration of a variable, except that now we decide about the visibility status of the new attribute. The resulting class transformer enriches a class named cl-ide by a new attribute at-ide.

```
add-abs-att : Identifier x TypExpDen x YokExpDen x PriSta ⟼ ClaTraDen      i.e.
add-abs-att : Identifier x TypExpDen x YokExpDen x PriSta ⟼ Identifier ⟼ WfState → WfState
add-abs-att.(at-ide, ted, yok, pst).cl-ide.sta =
   is-error.sta            ➔ sta
   declared.at-ide.sta     ➔ sta ◄ ‘attribute already declared’
   ted.sta : Error         ➔ sta ◄ ted.sta
   let
      ((cle, pre, cov), (obn, dep, ota, sft, ‘OK’))   = sta
   cle.cl-ide              ➔ ‘class unknown’
   let
      (cl-ide, tye, mee, obn)   = cle.cl-ide
      de-typ                    = ted.sta                      de-typ – declared type
      (tok, new-sft)            = get-tok.sft
      ref                       =
         pst = ‘public’  ➔ (tok, (de-typ, yok, $))
         pst = ‘private’➔ (tok, (de-typ, yok, cl-ide))
      new-cla = (cl-ide, tye, mee, obn[at-ide/ref])
   true                    ➔ ((cle[cl-ide/new-cla], pre, cov), (obn, dep, ota, new-sft, 'OK'))
```

Observe that in this definition we use the fact that the input state is well-formed, and therefore the internal name of a class, here cl-ide, is a component of this class.

Two more facts are to be noted that make abstract attribute declarations different from the declarations of variables (Sec. 6.7.2):

First, an attribute may be private, in which case its origin tag is equal to the name of the hosting class, rather than to $.

Second, if the declared type de-typ is an object type, in which case it is supposed to be a name of a class, we do not check if this is really the case, thus allowing to define abstract object-attributes with an *anticipatory referencing* to a class that hasn't been declared yet. This situation is illustrated by two examples written in Java (Fig. 6.7-1), where — additionally — we have to do with a class recursion.

class ListNode{
  int   no = 2;
  ListNode next;
   … }
	class A{
	 B b;
	 void b(B b){
	  this.b = b; } }
	class B{
	 A a = new A(); … }

Case A: simple recursion	Case B: mutual recursion

Fig. 6.7-1 Two examples of recursive anticipatory referencing in Java

In **Case A** class ListNode refers recursively to itself, in **Case B** class A refers to B, and B refers to A. Note that if in the second case class B would not refer to class A, then there would be no recursion but still an anticipatory referencing will be the case.

Since classes are regarded as types of objects, a question arises if we should allow anticipatory referencing between data types as well.

6.7.4.3 Adding a concrete attribute to the objecton of a class

add-con-att : Identifier x ValExpDen x TypExpDen x YokExpDen x PriSta ⟼ ClaTraDen i.e.

add-con-att : Identifier x ValExpDen x TypExpDen x YokExpDen x PriSta ⟼
⟼ Identifier ⟼ WfState → WfState

```
add-con-att.(at-ide, ved, ted, yok, pst).cl-ide.sta =
   is-error.sta            ➔ sta
   declared.at-ide.sta     ➔ sta ◄ 'attribute already declared'
   ted.sta : Error         ➔ sta ◄ ted.sta
   ved.sta = ?             ➔ ?
   ved.sta : Error         ➔ sta ◄ ved.sta
   let
      ((cle, pre, cov), (obn, dep, ota, sft, 'OK'))   = sta
   cle.cl-ide              ➔ 'class unknown'
   let
      (cl-ide, tye, mee, obn)  = cle.cl-ide
      de-typ                   = ted.sta                    de-typ – declared type
      de-val                   = ved.sta
      (tok, new-sft)           = get-tok.sft
      ref                      =
         pst = 'public'    ➔ (tok, (de-typ, yok, $))
         pst = 'private'   ➔ (tok, (de-typ, yok, cl-ide))
      new-cla = (cl-ide, tye, mee, obn[at-ide/ref])
   true ➔ ((cle[cl-ide/new-cla], pre, cov), (obn, dep[ref/de-val], ota, new-sft, 'OK'))
```

6.7.4.4 Concretizing abstract attributes and adding concrete attributes

Concretizations of an abstract attributes acts as assignments (Sec. 6.5.2) except that the concretized attribute must be abstract. In other words, we do not allow for a replacement of a value of an attribute by an new one at the stage of a class declaration (an engineering decision).

Adding a concrete attribute is a simple combination of adding an abstract attribute and concretizing it. We skip formal definitions of both constructors.

6.7.4.5 Adding a type constant to a class

The case of adding a type constant to a type environment includes three constructors (cf. Sec. 6.7.4.1): adding an abstract type, concretizing an abstract type and adding a concrete type. In the first case we add a type constant with a pseudotype Θ assigned to it.

```
add-abs-typ : Identifier ⟼ ClaTraDen
add-abs-typ : Identifier ⟼ Identifier ⟼WfState → WfState
add-abs-typ.ty-ide.cl-ide.sta =
   is-error.sta              ➔ sta
   declared.ty-ide.sta       ➔ sta ◄ ‘type name declared in state’
   let
      ((cle, pre, cov), sto) = sta
   cle.cl-ide = ?            ➔ sta ◄ ‘class unknown’
   let
      (cl-ide, tye, mee, obn)   = cle.cl-cla
      new-cla                   = (cl-ide, tye[ty-ide/Θ], mee, obn)
   true                      ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
```

In the second case we concretize an abstract type:

```
concretize-typ : Identifier x TypExpDen ⟼ ClaTraDen
concretize-typ : Identifier x TypExpDen ⟼ Identifier ⟼ WfState → WfState
concretize-typ.(ty-ide, ted).cl-ide.sta =
   is-error.sta       ➔ sta
      let
         ((cle, pre, cov), sto) = sta
      cle.cl-ide = ? ➔ sta ◄ ‘class unknown’
      let
         (cl-ide, tye, mee, obn)   = cle.cl-ide
   tye.ty-ide = ?     ➔ sta ◄ ‘type name unknown’
   tye.ty-ide ≠ Θ     ➔ sta ◄ ‘only abstract types may be concretized’
   ted.sta : Error    ➔ sta ◄ ted.sta
   let
      typ        = ted.sta
      new-cla    = (cl-ide, tye[ty-ide/typ], mee, obn)
   typ /: ObjTyp      ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
   cle.typ = ?        ➔ sta ◄ ‘object type unknown’
   true               ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
```

The following two conditions must be satisfied to concretize a type constant ty-ide:

1. ty-ide must be declared as an abstract type constant; i.e., we can’t change a type assigned to a concrete type constant, nor we can concretize a not declared constant,
2. if the type to be assigned to ty-ide is an object type, it must be the name of a declared class

The last case is analogous.

6.7.4.6 Adding a method constant to a class

In a method environment of a class we can bind three categories of pre-procedures — imperative and functional pre-procedures, and object pre-constructors — and three corresponding categories of signatures. We can also concretize abstract methods by completing signatures to pre-procedures. Since all the corresponding definitions are quite simple and analogous to each other, we show only three examples of such constructors. We start from introducing an auxiliary function:

get-parameters : ForParDen ↦ Sub.Identifier

which given a (list of) formal parameters returns the set of identifiers included in these parameters. We skip a formal definition of this function.

Our first constructor builds the denotation of a declaration of an imperative pre-procedure:

```
add-con-imp-met : Identifier x ImpProSigDen x ProDen ↦ ClaTraDen              i.e.
add-con-imp-met : Identifier x ForParDen x ForParDen x ProDen ↦
                                              ↦ Identifier → WfState → WfState
add-con-imp-met.(pr-ide, fpd-v, fpd-r, prd).cl-ide.sta =
   is-error.sta              ➔ sta
   let
      ((cle, pre, cov), sto) = sta
      (v-ide-1,…, v-ide-n)   = get-parameters.fpd-v
      (r-ide-1,…, r-ide-k)   = get-parameters.fpd-r
   cle.cl-ide = ?            ➔ sta ◄ 'class unknown'
   declared.pr-ide.sta       ➔ sta ◄ 'identifier not free'
   declared.v-ide-i.sta      ➔ sta ◄ 'identifier not free'     for i = 1;n[54]
   declared.r-ide-i.sta      ➔ sta ◄ 'identifier not free'     for i = 1;k
   let
      (cl-ide, tye, mee, obn)   = cle.cl-ide
   let
      ipp        = create-imp-pre-pro.(fpd-v, fpd-r, prd, cl-ide)
      new-cla    = (ide, tye, mee[pr-ide/ipp], obn)
   true                      ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
```

The second constructor corresponds to concretizing a previously declared functional method:

```
concretize-fun-met : Identifier x FunProSigDen x ProDen x ValExpDen   ↦ClaTraDen
concretize-fun-met : Identifier x FunProSigDen x ProDen x ValExpDen ↦
                                              ↦ Identifier → WfState → WfState
concretize-fun-met.(pr-ide, fps, prd, ved).cl-ide.sta =
   is-error.sta                   ➔ sta
   let
      ((cle, pre, cov), sto) = sta
   cle.cl-ide = ?                 ➔ sta ◄ 'class unknown'
   let
      (cl-ide, tye, mee, obn)   = cle.cl-ide
   mee.pr-ide = ?                 ➔ sta ◄ 'method unknown'
   mee.pr-ide /: FunProSigDen     ➔ sta ◄ 'signature of functional procedure expected'
   fps ≠ mee.pr-ide               ➔ sta ◄ 'signatures not compatible'
   let
      fpp        = create-fun-pre-pro.(fps, prd, ved, cl-ide)
      new-cla    = (cl-ide, tye, mee[pr-ide/fpp], obn)
   true                           ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
```

---

[54] Denotationally it is not necessary that formal parameters are free, but we take this assumption since it technically simplifies future rules of correct program development (cf. Sec. 9.4.4.2).

Third constructor corresponds to a declaration of a concrete object constructor.

```
add-con-obj-met : Identifier x ObjConSigDen x ProDen ⟼ ClaTraDen
add-con-obj-met : Identifier x ForParDen x Identifier x ProDen ⟼
                                    ⟼ Identifier → WfState → WfState
add-con-obj-met.(oc-ide, fpd, cl-ide, prd).cl-ide.sta =          oc- object-constructor
  is-error.sta             ➔ sta
  let
    ((cle, pre, cov), sto) = sta
  cle.cl-ide = ?           ➔ sta ◄ 'class unknown'
  let
    (cl-ide, tye, mee, obn)  = cle.cl-ide
  declared.oc-ide.sta      ➔ 'identifier not free'
  let
    opc        = create-obj-pre-con.((fpd, cl-ide), prd)
    new-cla    = (cl-ide, tye, mee[oc-ide/opc], obn)
  true                     ➔ ((cle[cl-ide/new-cla], pre, cov), sto)
```

6.7.4.7 Composing transformers sequentially

The following constructor simply combines the "declaration layers" of transformers sequentially:

```
compose-cla-tra : ClaTraDen x ClaTraDen ⟼ ClaTraDen
compose-cla-tra.(cdt-1, cdt-2).ide = (ctr-1.ide) ● (ctr-2.ide)
```

Intuitively speaking a sequential composition of transformers decribes a process of a cumulative creation of a class named ide, provided that it is declared in the argument state. In this process a class asigned ot ide (if any) is modified to a new class by adding to it some new items. Note that this process is possible only "internally" within a class declaration, since this is the only context in which we can use class transformers. It can't be performed "externally" since class transformers do not belong to the category of declarations. Consequently, a class once declared, can't be changed in the future.

Note also that compose-cla-tra is associative, since ● is associative.

6.7.5 Enrichments of covering relations

Our last category of declarations are enrichments of covering relations. Although they refer to types, that are "stored" in classes, the enrichments of covering relations do not transform classes, but environments. This is an engineering decision which makes covering relations globally accessible.

```
enrich-cov : TypExpDen x TypExpDen  ⟼ DecDen                     i.e.
enrich-cov : TypExpDen x TypExpDen  ⟼ WfState ⟼ WfState
enrich-cov.(ted-1, ted-2).sta =
  is-error.sta            ➔ sta
  ted-i.sta : Error       ➔ sta ◄ ted-i                                   for i = 1,2
  let
    typ-i                                  = ted-i.sta                    for i = 1,2
    ((cle, pre, cov), (obn, dep, ota, sft, 'OK'))  = sta
    cov-1                                  = enrich-cov.(cov, typ-1, typ-2)
  cov-1 : Error           ➔ sta ◄ cov-1
  typ-1, typ-2 /: ObjTyp  ➔ ((cle, pre, cov-1), (obn, dep, ota, sft, 'OK'))
  true                    ➔
    let
      ide-i  = typ-i                                                      for i = 1,2
    cle.ide-i = ?   ➔ 'object types must point to declared classes'       for i = 1,2
    true            ➔ ((cle, pre, cov-1), (obn, dep, ota, sft, 'OK'))
```

We recall that enrich-cov (Sec. 5.4.2) realizes the requirement that if one of the types is an object type, then so must be the other. Additionally our constructor checks if object types point to declared classes.

A word of comment is necessary here to explain why the enrichments of covering relations were included in the category of declarations rather than instructions. The reason is to ensure that once program execution passes the declaration part of the program (cf. Sec. 6.3), the covering relation won't be changed. This assumption simplifies the future rule of creating correct procedure calls (see Sec. 9.4.6.3).

6.7.6 The openings of procedures

As we have assumed in Sec. 6.3 every program in our language is a sequential combination of three components:

1. a declaration, possibly composed,
2. a single predefined *procedures' opening*,
3. an instruction, also possibly composed

The assumption 2. means that the domain of procedures' openings includes only one element

pod : ProOpeDen = {open-pro-den}

where

open-pro-den : WfState ⟼ WfState,

and that this element is element is built by a zero-argument constructor

create-open-pro-den : ⟼ ProOpeDen

i.e. it is built by language designer, rather than by programmers, as it is the case with declarations and instructions. To incorporate opening declarations into our model, we first introduce an auxiliary function

get-pre-pro : WfState ⟼ (ProIndicator x PrePro)^{c*} get pre-procedures

This function given a state ((cle, pre, cov), sto), returns a sequence of all pairs ((cl-ide, pr-ide), prp) where:

- pr-ide is a name of a pre-procedure declared in a class named cl-ide,
- prp is the corresponding pre-procedure.

We skip a formal definition of this function, and we assume that the pairs of identifiers (cl-ide, pr-ide) will be called *procedure indicators*. Now, a half-formal definition of our constructor is the following:

```
create-open-pro-den : ⟼ ProOpeDen
create-open-pro-den : ⟼ WfState ⟼ WfState
create-open-pro-den .().dt-sta =                              dt-sta declaration-time state
   is-error.dt-sta               ➔ dt-sta
   get-pre-pro.dt-sta = ()       ➔ dt-sta ◄ 'no procedures to declare'
   let
      ((pri-1, prp-1),…,(pri-n, prp-n))  = get-pre-pro.dt-sta
      ((dt-cle, dt-pre, dt-cov), dt-sto) = dt-sta
      pro-1                              = prp-1.(dt-cle, dt-pre[pri-1/pro-1,…, (pri-n/pro-n], dt-cov)
      …
      pro-n                              = prp-n.(dt-cle, dt-pre[pri-1/pro-1,…, (pri-n/pro-n], dt-cov)
      (dt-cle, dt-pre[pri-1/pro-1,…, (pri-n/pro-n]), dt-cov) = ot-env      open-time environment
   true                          ➔ (ot-env, dt-sto)
```

This constructor given an empty tuple of arguments returns a state-to-state function open-pro-den. This function gets a *declaration-time state* dt-sta, and generates a tuple of procedures as a least solution of a set of fixed-point equations that refer to pre-procedures declared in the classes of dt-sta. These procedures are then assigned to the corresponding procedure indicators in the declaration-time environment, thus creating an

open-time environment. This environment together with the declaration-time store, creates an *open-time state*.

Note that if we execute a "main program", i.e., a program which is not a procedure body in a procedure call, then the declaration-time procedure environment dt-pre is empty.

It is worth observing in this place that in our definition we do not build any stack mechanism usually engaged associated with recursion by interpreters or compilers. We do not need to do so, since we describe the recursion in **Lingua** using the recursion in **MetaSoft**.

Now, let's try to make the definition of our constructor a little more formal. To do this we first define a family of metaconstructors MC[n] indexed by positive integers n:

```
MC[n] : (ProIndicator x PreProc)^cn x State ⟼ Procedure^cn ⟼ Procedure^cn
MC[n].(((pre-1, prp-1),…,(pri-n, prp-n)), sta).(pro-1,…,pro-n) =
   let
      ((cle, pre, cov), sto)   = sta
            new-pro-1 = prp-1.(dt-cle, dt-pre[pri-1/pro-1,…, (pri-n/pro-n])
               …
            new-pro-n = prp-n.(dt-cle, dt-pre[pri-1/pro-1,…, (pri-n/pro-n])
   true ➔ (new-pro-1,…,new-pro-n)
```

Then by

LFP : (A ⟼ A) ⟼ A

we denote a universal function such that if A is a CPO (Sec. 2.4), and if F : A ⟼ A is a continuous function in this CPO, then LFP.F is the least fixed point of F. With these metafunctions we can write our definition in the following way:

```
create-pro-opening : ⟼ DecDen
create-pro-opening : ⟼ WfState ⟼ WfState
create-pro-opening.().dt-sta =                                   dt-sta a declaration-time state
   is-error.dt-sta                 ➔ dt-sta
   get-pre-pro.dt-sta = ()         ➔ dt-sta
   let
      ((dt-cle, dt-pre), dt-sto)              = dt-sta
      ((pri-1, prp-1),…,(pri-n, prp-n))       = get-pre-pro.dt-sta
      (pro-1,…,pro-n)                         = LFP.(MC[n].(((pri-1, prp-1),…,(pri-n, prp-n))
   true                            ➔ ((dt-cle, dt-pre[pri-1/pro-1,…, pri-n/pro-n]),sto)
```

Of course, Procedure^{cn} is a CPO with a componentwise ordering and Procedure is ordered by a set-theoretical inclusion of functions. It remains to be proved that

MC[n].(((pri-1, prp-1),…,(pri-n, prp-n)), sta))

is a continuous function in Procedure^{cn}.

For technical reasons we introduce a constructor of a *trivial opening*, that is an identity function :

open-skip : ⟼ ProOpeDen.

7 SYNTAX AND SEMANTICS

7.1 An overview of syntax derivation

The derivation of a syntax in our model starts from the signature SigAlgDen[55] of the algebra of denotations and proceeds in three steps corresponding to three transformations:

S2A : SigAlgDen ⟼ **AlgAbsSyn** the creation of an algebra of abstract syntax[56]
A2C : **AlgAbsSyn** ⟼ **AlgConSyn** the transformation of abstract syntax into concrete syntax
C2C : **AlgConSyn** ⟼ **AlgColSyn** the transformation of concrete syntax into colloquial syntax

Each of these transformations is a many-sorted function, and A2C is (additionally) a homomorphism. We build our algebras in such a way that for each of them there exists a corresponding many-sorted function of semantics:

A2D : **AlgAbsSyn** ⟼ **AlgDen** abstract semantics
C2D : **AlgConSyn** ⟼ **AlgDen** concrete semantics
SEM : **AlgColSyn** ⟼ **AlgDen** colloquial semantics

The first two semantics are homomorphisms, i.e., are denotational semantics, whereas the third one is not. Since colloquial syntax will be the ultimate user-syntax of our language, its semantics will be called *the semantics of* **Lingua**.

All syntactic algebras will be described by corresponding equational grammars (Sec. 2.15). These grammars explicitly define the carriers of our algebras, and implicitly — i.e., by grammatical clauses — their constructors. For instance, the following grammatical equation:

cre : ConRefExp =
  ref (Identifier) |
  ref ConValExp **at** Identifier **fer**

defines the carrier of concrete reference expressions, and each line of this equation below the sign =, called a *grammatical clause*, defines a corresponding constructor of the algebra of concrete syntax:

con-ref-variable.ide = **ref** (ide)
con-ref-attribute.(cve, ide) = **ref** cve **at** ide **fer**

Abstract-syntax grammar is always LL(k) (see Sec. 2.16) which makes abstract-syntax programs easily parsable[57]. The derivation of this grammar can be made algorithmic.

Since abstract syntax is usually awkward to use, in the next step we build an *algebra of concrete syntax* which is, by the rule, a homomorphic image of the abstract-syntax algebra:

A2C : **AlgAbsSyn** ⟼ **AlgConSyn**

This step is not algorithmic, since here we take major decisions about the future shape of our syntax, and we try to make it possibly user friendly. In building concrete syntax, we must ensure that the corresponding concrete semantics exists. For this to be the case, A2C must be adequate (Sec. 2.14), which means that it must not glue more than A2D. If it is so, the concrete semantics is unique and satisfies the equation

---

[55] This metavariable is not typeset in bold since a signature of an algebra is not an algebra itself.

[56] S2A is read as “signature to abstract”, and the remaining symbols are read analogously.

[57] In fact, abstract syntax scripts may be regarded as linear representations of parsing trees.

C2D = A2C^{-1} ● A2D

where A2C^{-1} denotes a chosen inverse of A2C, and corresponds to a parsing step from concrete to abstract syntax. Of course, if A2C is not an isomorphism, then there is more than one parsing procedures "reversing" A2C, and therefore A2C^{-1} should be regarded as just one of them.

Although the majority of grammatical clauses of concrete syntax can be made user-friendly, a few of them may require further modifications. As a rule these modifications are not homomorphic, since, if they were, they could have been included in A2C. These modifications lead us to a colloquial-syntax grammar, and to the corresponding algebra. In this case we make sure that there exists a many-sorted function

RES : **AlgColSyn** ⟼ **AlgConSyn**

that we call a *restoring transformation*. It restores colloquial syntax "back to" the concrete one. Now, the semantics of our language may be regarded as a composition of two many-sorted functions:

SEM = RES ● C2D

This semantics, is no more denotational, since it is not a homomorphism. Still, as we are going to see in Sec. 7.5.1, it may be said to be "denotational to a large extent".

In the following sections we describe equational grammars of our three syntaxes. For the sake of brevity we shall not list all clauses of these grammars, but only their typical examples. The used notation has been described in Sec. 2.15. The components of our many-sorted functions will be indexed by suffixes indicating the corresponding carriers of algebras. E.g.

A2C.ins: AbsIns ⟼ ConIns

is a component of A2C that corresponds to instructions.

7.2 Abstract syntax

7.2.1 General remarks

As a rule abstract syntax is a prefixed syntax which means that each syntactic element starts from a prefix that is (not quite formally) an Arial Narrow copy of an Arial metaname of the corresponding constructor of denotations.

7.2.2 Identifiers, class indicators and privacy statuses

In this category we have three grammatical equations (see Sec. 6.2):

```
ide  : Identifier = …
cli  : ClaInd     = empty-class | Identifier
pst  : PriSta     = private | public
```

7.2.3 Type expressions

```
ate  : AbsTypExp =
        ted-create-bo() | ted-create-in.()  …                  |
        ted-constant( AbsIdentifier , AbsIdentifier )          |
        ted-create-li( AbsTypExp )                             |
        …
```

7.2.4 Value expressions

We assume to be given (i.e. somehow defined as parameters of our model) four auxiliary syntactic domains. We do not call them "abstract syntactic", since they will be common for all three syntaxes.

```
BooleanSyn  = {true, false}
IntegerSyn  = …
RealSyn     = …
TextSyn     = …
```

The elements of these domains are symbols or strings of symbols representing corresponding elements of denotational domains. Examples of syntactic representations of integers or reals may be: 432894713984713847 for an integer or 9874,0951208515584958490 for a real number. The grammatical equation corresponding to value expression is the following:

```
ave : AbsValExp =
        ved-boo(BooleanSyn)                                      |
        ved-int(IntegerSyn)                                      |
        ved-rea(RealSyn)                                         |
        ved-tex(TextSyn)                                         |
        ved-variable(AbsIdentifier)                              |
        ved-attribute(AbsValExp , AbsIdentifier)                 |
        ved-call-fun-pro(AbsIdentifier, AbsIdentifier, ActParAbs) |
        ved-divide-re(AbsValExp , AbsValExp)                     |
        ved-equal(AbsValExp , AbsValExp)                         |
        ved-or(AbsValExp , AbsValExp)                            |
        ved-create-li(AbsValExp)                                 |
        ved-get-from-rc(AbsValExp , AbsIdentifier)               |
        …
```

7.2.5 Reference expressions

```
are : AbsRefExp =
        ref-variable(Identifier)                |     a reference of a variable
        ref-attribute(AbsValExp , AbsIdentifier)      a reference of an object attribute
```

7.2.6 Yoke expressions

```
ate : AbsYokExp    =
        yo-pass()                               |
        yo-sum-in()                             |
        yo-give-td(AbsValExp)                   |

        yo-add-in(AbsYokExp , AbsYokExp)        |
        …
        yo-top(AbsYokExp)                       |
        yo-get-from-ar(AbsValExp)               |
        yo-get-from-re(AbsYokExp , Identifier)  |

        yo-equal-in(AbsYokExp , AbsYokExp)      |
        yo-less-in(AbsYokExp , AbsYokExp)       |
        yo-no-rep-in(AbsYokExp)                 |
        yo-increasing-in(AbsYokExp)             |

        yo-true()                               |
        yo-and(AbsYokExp , AbsYokExp)           |
        yo-or(AbsYokExp , AbsYokExp)            |
        yo-not(AbsYokExp)                       |

        yok-all-of-li(AbsYokExp)                |     general quantification
        yok-exists-in-li(AbsYokExp)             |     existential quantification
        …
```

7.2.7 Instructions

```
ain : AbsIns =
        assign(AbsRefExp , AbsValExp)                                        |
        enrich(AbsTypExp , AbsTypExp)                                        |
        call-imp-pro(AbsIdentifier , AbsIdentifier , ActParAbs , ActParAbs)  |
        call-obj-con(AbsIdentifier , AbsIdentifier , ActParAbs)              |
        skip-ins()                                                           |
        if(AbsValExp , AbsIns , AbsIns)                                      |
        if-error(AbsValExp , AbsIns)                                         |
        while(AbsValExp , AbsIns)                                            |
        compose-ins(AbsIns , AbsIns)
```

7.2.8 Declarations

```
ade : AbsDec =
        var-dec(AbsListOfIde , AbsTypExp , AbsYokExp)     |
        enrich-cov-rel(AbsTypExp , AbsTypExp)             |
        cla-dec(AbsIdentifier , AbsClaExp , AbsClaTra)    |
        compose-dec(AbsDec , AbsDec)                      |
        skip-dec()
```

7.2.9 Openings of procedures

```
aop : AbsOpePro = create-open-pro()
```

7.2.10 Class transformers

```
act : AbsClaTra =
        add-abs-att(AbsIdentifier , AbsTypExp , AbsYokExp, PriSta)            |
        ...
        add-con-imp-met(AbsIdentifier , AbsImpProSig , AbsPro)                |
        add-con-fun-met(AbsIdentifier , AbsFunProSig , AbsPro, AbsValExp)     |
        add-con-obj-con(AbsIdentifier , AbsObjConSig , AbsPro)                |
        ...
        compose-cla-tra(AbsClaTra, AbsClaTra)
```

7.2.11 Preambles of programs

```
app : AbsProPre =
        make-ppd-of-dcd(AbsDec)          |
        make-ppd-of-ind(AbsIns)          |
        compose(AbsProPre , AbsProPre)
```

7.2.12 Programs

```
apr : AbsPro = make-prog( AbsProPre , AbsOpePro , AbsIns )
```

7.2.13 Declaration-oriented carriers

```
ali : AbsLisOfIde   =
        build-loi(AbsIdentifier)          |
```

add-to-loi(Identifier , ListOfIde)

ads : AbsDecSec =
build-dse(AbsLisOfIde , AbsTypExp)

afp : AbsForPar =
build-fpd(AbsDecSec) |
add-to-fpd(AbsDecSec , AbsForPar)

aap : AbsActPar =
build-apd(AbsLisOfIde)

Intuitively the last equation means that abstract actual parameters are just list of identifiers. Set-theoretically we could have dropped the category AbsActPar, and use AbsLisOfIde instead, but algebraically we keep it formally have the concept of actual parameters in our model.

7.2.14 Signatures

ais : AbsImpProSig =
build-ipsd(AbsForPar , AbsForPar)

afs : AbsFunProSig =
build-fpsd(AbsForPar , AbsTypExp)

aos : AbsObjConSig =
build-ocsd(AbsForPar , AbsIdentifier)

7.3 Concrete syntax

7.3.1 General remarks

In the abstract-to-concrete step, we modify syntax to make it more user-friendly but keep its semantics denotational. Technically, our modifications will belong to four categories:

1. the simplification of prefixes,
2. the omission of prefixes but keeping parenthesizing,
3. the modifications from prefix to infix notation,
4. the omission of parentheses wherever possible without violating semantics' homomorphicity.

The first three categories include transformations that are isomorphic-like, i.e. unambiguously reversible. The fourth group makes A2C not isomorphic, but still homomorphic. In this step, we go only as far, as the adequacy of A2C permits (cf. Sec. 2.14), i.e. we omit parentheses corresponding to sequential composition of declarations and instructions, but not to arithmetic operations. In this step we sacrify LL(k)-ness for frendliness.

In the end we have to emphasise that our discussion of the abstract-to-concrete step is very sketchy. We only try to show a general potential of this step, leaving its further elaboration and analysis to a future research.

7.3.2 Identifiers, class indicators and privacy statuses

ide : Identifier = …
cli : ClaInd = empty-class | Identifier
pst : PriSta = private | public

7.3.3 Type expressions

```
cte   : ConTypExp =
        boolean | integer  …                     |
        constant( Identifier , Identifier )      |
        list-of(ConTypExp )                      |
        …
```

In the two first clauses we only shorten prefixes, which is an isomorphic-like transformation. In the second clause we allow writing MyClass.myType instead of ted-constant(MyClass, myType) which is again an isomorphic transformation, although now the proof of this fact may be not trivial. Note that a script of the form ide-1.ide-2 may be a type constant or a value expression, and we have to see the context of this script to identify which one it is.

7.3.4 Value expressions

We assume to be given the same auxiliary domains as in Sec. 7.2.3. The grammatical equation defining concrete value expressions is the following:

```
cve : ConValExp =
        true | false                           |
        IntegerSyn                             |  syntactic representations of integers
        RealSyn                                |  syntactic representations of reals
        ‘ TextSyn ‘                            |  texts are closed in apostrophes
        Identifier                             |  variable
        Identifier . Identifier                |  getting an attribute of an object
        call Identifier.Identifier(ConActPar)  |  calling an imperative procedure
        (ConValExp /. ConValExp)               |  real division is a “division with dot /.
        (ConValExp = ConValExp)                |  equality
        (ConValExp or ConValExp)               |  disjunction
        make list(ConValExp)                   |  making a one-element list
        rec ConValExp at Identifier cer        |  getting an attribute of a record
        …
```

Here we switch from prefix- to infix notation, but we keep the parentheses structures, although the symbols of parentheses may change, e.g. from “mathematical” ones like (and) to program oriented like **rec** and **cer**. An example of a clause of the definition of A2C.cve may be:

```
A2C.cve.[ ved-call-fun-pro(aid-c , aid-p , apa) ] =          -c for “class, -p for “procedure”
   call A2C.cid.[aid-c] . A2C.cid.[aid-p] ( A2C.apa.[apa] )
```

7.3.5 Reference expressions

```
cre : ConRefExp =
        ref (Identifier)                       |
        ref ConValExp at Identifier fer
```

7.3.6 Yoke expressions

```
cyo   : ConYokExp =
        value                                  |
        sum-in                                 |
        ConValExp                              |
        (ConYokExp + ConYokExp)                |
        …
```

```
top(ConYokExp)                          |
array[ConValExp]                        |
record.Identifier                       |

(ConYokExp = ConYokExp)                 |
(ConYokExp < ConYokExp)                 |
yo-no-rep-in(ConYokExp)                 |
yo-increasing-in(ConYokExp)             |

TT                                      |
(ConYokExp and ConYokExp)               |
(ConYokExp or ConYokExp)                |
not(ConYokExp)                          |

all-in-arr(ConYokExp)                   |
exists-in-arr(ConYokExp)                |
...
```

Note that the resignation of some prefixes makes our grammar not LL(k). At the same time, however, we keep parentheses to protect the adequacy of our homomorphism (Sec. 2.14). As we are going to see in Sec. Sec. 7.4 and 7.5, the omission of some parentheses may lead to a "not quite denotational" semantics.

7.3.7 Instructions

```
cin : ConIns =
    ConRefExp := ConValExp                                   |
    enrich( ConTypExp, ConTypExp )                           |
    call Identifier.Identifier ( val ConActPar ref ConActPar ) |
    new Identifier by Identifier.Identifier (ConActPar)      |  calls of object constr.
    while ConValExp do ConIns od                             |
    if ConValExp then ConIns else ConIns fi                  |
    skip-ins                                                 |
    ConIns ; ConIns
```

In the first clause we changed prefix notation to infix notation, and we skipped parentheses. The latter transformation is not harmful for the adequacy of the homomorphism, since — intuitively — assignment instructions will be always closed by the parentheses of "other structures" such as **;**, **do**, **then**, etc. A formal proof should be carried by induction on the structure of our grammar.

In the last clause we have also skipped parentheses, but in this case to prove that such a transformation does not destroy the adequacy of our homomorphism we have to use the fact that the sequential composition of functions is associative, and refer to Theorem 2.14-1 in Sec 2.14.

7.3.8 Declarations

```
cde : ConDec =
    let ConLisOfIde be ConTypExp with ConYokExp tel          |
    enrich-cov( ConTypExp, ConTypExp )                       |
    class Identifier parent ConClaExp with ConClaTra ssalc   |
    skip-dec                                                 |
    ConDec ; ConDec
```

Similar comments, as in Sec. 7.3.7, apply here to the last clause.

7.3.9 Openings of procedures

```
cpo : ConProOpe = open procedures
```

7.3.10 Class transformers

```
cct : ConClaTra =
        let Identifier be ConTypExp with ConYokExp as PriSta tel          | add abs. attribute
        ...
        set Identifier be ConTypExp tes                                   | add con. typ const.
        proc Identifier (ConImpProSig) begin ConPro end                   |
        fun Identifier (ConFunProSig) begin ConPro return ConValExp end   |
        obj Identifier (ConObjConSig) begin ConPro end                    |
        …
        ConClaTra ; ConClaTra                                             |
        skip-ctr
```

The following component of procedure declaration:

```
Identifier (ConImpProSig)
```

will be called a *procedure header*.

7.3.11 Preambles of programs

```
cpp  ConProPre =
        ConDec                |
        ConIns                |
        ConProPre ; ConProPre
```

7.3.12 Programs

```
cpr : ConPro = ConProPre ; open procedures ; ConIns
```

Here we may safely drop parentheses since they are the outermost parentheses in a program, and therefore their removal do not destroy isomorphicity.

7.3.13 Declaration-oriented carriers

```
cli : ConLisOfIde   =
        Identifier              |
        Identifier , ConLisOfIde

cds : ConDecSec  =
        ConLisOfIde as ConTypExp

cfp : ConForPar =
        ConDecSec               |
        ConDecSec , ConForPar

cap : ConActPar =
        ConLisOfIde
```

See a comment about the last equation in Sec. 7.2.13.

7.3.14 Signatures

```
cis   : ConImpProSig    = val ConForPar ref ConForPar
cfs   : ConFunProSig    = val ConForPar ret ConTypExp
col   : ConObjConSig    = val ConForPar ret Identifier
```

An example of a concrete imperative-procedure declaration may be the following

```
proc compute ( val x, y as integer, z as real,
               ref p, r as integer )
               begin
                  cpr
               end
```

where cpr is a concrete program.

7.4 Colloquial syntax

Colloquial syntax is, as a rule, THE syntax of our language, i.e., the syntax to be used by programmers. Consequently the metavariables (non-terminals) of colloquial syntax will not be prefixed by Col. E.g., instead of writing ColValExp we shall write just ValExp.

In our language we introduce two categories of colloquialisms: the omission of parentheses in arithmetic and boolean expression, and the creation of a new constructor of attribute declarations.

The omission of parentheses in arithmetic and boolean expressions concerns value expressions and yokes. Formally this means that in a concrete-to-colloquial step to every parenthesized colloquial clause such as, e.g.,

```
(ValExp +. ValExp)
```

we add a corresponding parentheses-free clause

```
ValExp +. ValExp
```

Consequently the grammatical equation for colloquial value-expressions will look as follows:

```
vex : ValExp =
        true | false                       |
        IntegerSyn                         |
        RealSyn                            |
        ‘ TextSyn ‘                        |
        Identifier                         |
        obj ValExp at Identifier jbo       |
        call Identifier.Identifier(ActPar) |
        (ValExp /. ValExp)                 |
        ValExp /. ValExp                   |   new clause (without parentheses)
        …
```

In colloquial syntax parentheses are optional — we may use them or not. The signature of the algebra of colloquial syntax is, therefore, an extension of the signature of concrete syntax by constructors building parentheses-free expressions.

The corresponding restoring transformation for arithmetic expressions adds parentheses according to the rule that multiplication and division bind stronger than addition and subtraction, and the “remaining” parentheses are added from left to write. E.g., the colloquial expression:

```
a + b*c – e*f
```

will be restored to

```
((a + (b*c)) – (e*f)).
```

7.4.1 New constructor of attribute declarations

In the repertuar of class transformers (Sec. 6.7.4) we have only one constructor that concerns attributes, namely the declaration of an abstract attribute. Since the concretization of an abstract attribute may be realized

by an assignment instruction, we have not introduced at this level a constructopr that adds a concrete attribute. Now, to allow programmers to declare a concrete attribute in one step, e.g. :

let abscissa = 2,15 **be** real **and** public **tel**

we introduce the following colloquial-grammar clause :

let Identifier = ValExp **be** TypExp **and** PriSta **tel**

and we assume that such colloquial declarations are restored to the following concrete forms

let Identifier **be** TypExp **and** PriSta **tel**;
Identifier **:=** ValExp

Of course, we could have introduced this constructors at the level of denotations, but instead we decided to do it at the level of colloquial syntax just to show that at the level of denotations we do not necessarily need to care about the future colloquial syntax. When we design a programming language we may take our decisions about syntax as late as possible.

7.4.2 The list of colloquial domains

Since we are going to use our colloquial syntax in the investigations about validating programming in Sec. 9, we list below all colloquial-syntax domains and their corresponding metavariables. Since colloquial syntax is the ultimate syntax of the user, we do not prefix the names of colloquial domains with Col, analogously to Abs and Con. E.g. instead of talking about “colloquial expressions” we shall talk about “expressions”.

ide : Identifier — identifiers
cli : ClaInd — class indicators
pst : PriSta — privacy statuses
tex : TypExp — type expressions
yex : YokExp — yoke expressions
vex : ValExp — value expressions
rex : RefExp — reference expressions
ins : Instruction — instructions
dec : Declaration — declarations
opp : OpePro — the opening of procedures
ctr : ClaTra — class transformers
ppr : ProPre — program preambles
prg : Program — programs
loi : LisOfIde — lists of identifiers
des : DecSec — declaration sections
fpa : ForPar — formal parameters
apa : ActPar — actual parameters
ips : ImpProSig — imperative-procedure signatures
fps : FunProSig — functional-procedure signatures
ocs : ObjConSig — object-constructor signatures

7.5 Semantics

7.5.1 The ultimate semantics of Lingua

Since in our model colloquial syntax is assumed to be the user’s “ultimate” syntax, its semantics:

SEM : **AlgColSyn** $\mapsto$ **AlgDen**

will be called just *the semantics* of **Lingua**. It is not a homomorphism, and symbolically may be written as a composition of three many-sorted mappings

SEM = RES ● A2C^{-1} ● A2D

where:

- RES is a restoring transformation from colloquial syntax to concrete syntax,
- A2C^{-1} is a chosen invers of A2C and represents a parsing step,
- A2D is a homomorphism which constitutes the semantics of abstract syntax.

It is to be emphasized in this place that whereas the choice of A2C^{-1} is irrelevant for SEM, the choice of RES defines the way in which we understand colloquial syntax.

Let us start the process of building the definition of semantics from the definition of A2D. This is an easy step, since the grammar of abstract syntax is unambiguous. In this step for each syntactic category, i.e., for each carrier of the algebra of abstract syntax, we create one definitional equation with several *semantical clauses*. E.g., for the category of programs the equation includes only one clause, and is the following:

```
A2Dapr.[apr] =
   apr :: make-prog( ade , create-pro-opening() , ain )  ➔
      make-prog.(A2D.ade.[ade], create-pro-opening(), A2D.ain.[ain]
```

In the first line of this definition, the first apr is an index, and the apr in square brackets is a metavariable running over AbsPro.

The second and the third line constitute together one semantical clause. The symbol :: denotes (not quite formally) a pattern matching operator, and expresses the fact that apr is parsable to the form make-prog(ade , create-pro-opening() , ain). Since our grammar is unambiguous, this parsing is unique.

The constructor of denotations make-prog is the semantic counterpart of the prefix make-prog. This constructor is applied to the denotations of the components of the program, which expresses the compositionality (denotationality) of the semantics or abstract syntax. Another typical example, which this time concerns instructions, is the following:

```
A2Dain.[ain] =
   ain :: assign(are , ave)                          ➔
      assign.(A2Dare.[are], A2Dave.[ave])
   ain :: call-imp-pro(aid-1.aid-2 , apa-1 , apa-2) ➔
      call-imp-pro.( A2Daid.[aid-1] , A2Daid.[aid-2] , A2Dapa.[apa-1] , A2Dapa.[apa-2])
   …
   ain :: seq-ins(ain-1 , ain-2)                     ➔
      seq-ins.(A2Dain.[ain-1], A2Dain.[ain-2])
```

Now, let's pass to the definition of A2C^{-1}. Since A2C is not an isomorphism, we have to choose one of alternative parsing strategies, but since it is adequate, this choice is irrelevant for the "meaning" of C2D. Note also that A2C introduces only "three ambiguities", namely the omissions of parentheses in three grammatical concrete clauses:

```
(ConIns ; ConIns)
(ConDec ; ConDec)
(ConClaTra ; ConClaTra)
```

Since these cases are similar to each other, let's analyze the case of instructions. To define a chosen parsing strategy — let's call it C2A — we introduce an auxiliary subdomain of instructions called *atomic instructions*:

```
atin : AtomIns =
        ConRefExp := ConValExp                                    |
        call Identifier.Identifier ( val ConActPar ref ConActPar ) |
        new Identifier by Identifier.Identifier (ConActPar)       |
        while ConValExp do ConIns od                              |
        if ConValExp then ConIns else ConIns fi                   |
        skip-ins
```

It is now easy to prove (by induction) that every concrete non-atomic instruction is unambiguously parsable into an instruction of the form:

atin ; cin,

where cin may, of course, include some nonatomic instructions.

We emphasize that AtomIns is not regarded as a new carrier of the algebra of concrete syntax, but as an auxiliary domain "outside" of this algebra. We may say that we do not modify the algebra of concrete syntax, but we build an auxiliary one, to be used only for the purpose of parsing[58].

Now, the definition of C2A.cin may be written as follows

```
C2Acin.[cin] =
   cin :: cre := cve                              →
      assign(C2Acre.[cre] , C2Acve.[cve])
   cin :: call cid-c.cid-p ( val cap-v ref cap-r)  →
      call-imp-pro(C2Acid.[cid-c] , C2Acid.[cid-p] , C2Acap.[cap-v] , C2Acap.[cap-r])
   …
   cin :: atin ; cin                              → seq-ins(C2Acin.[atin] , C2Acin.[cin])
```

From this definition, the definition of A2D, and the equation

$$\text{C2D} = \text{C2A} \bullet \text{A2D} \tag{7.5-1}$$

we can algorithmically generate the following equation of the definition of C2D:

```
C2Dcin.[cin] =                                                              (7.5-2)
   cin :: cre := cve                              →
      assign.(C2Dcre.[cre], C2Dcve.[cve])
   cin :: call cid-1.cid-2 ( val cap-1 ref cap-2) →
      call-imp-pro.(C2Dcid.[cid-1] , C2Dcid.[cid-2] , C2Dcap.[cap-1] , A2Dcap.[cap-2])
   …
   cin :: atin ; cin                              →
      seq-ins.(C2Dcin.[atin], C2Dcin.[cin])
```

Let's see how it works for the first clause:

```
C2Dcin.[cre := cve]                                    =                     by (7.5-1)
A2Dain.[C2Aain.[ cre := cve ]]                         =       by isomorphicity of C2A
A2Dain.[ assign(C2Acre.[cre] , C2Acve.[cve]) ]         =    by homomorphicity of A2D
assign.( A2Dare.[ C2Acre.[cre] ], A2Dave.[C2Acve.[cve] ] )  =                by (7.5-1)
assign.( C2Dcre.[cre], C2Dcve.[cve] )
```

In the second transformation we use the fact that in the case of assignments, A2Cain hence also C2Acin, are reversible, i.e. "locally isomorphic".

Let us pass now to the restoring function RES. Similarly as C2A, also RES adds parentheses, but now, the way it does it makes difference for the meaning of expressions. E.g., if we decide to add the "missing" parentheses to

a + b * c + (d – e) * f

in assuming that multiplication bind stronger than addition, and besides we add them from left to right:

((a + (b * c)) + ((d – e)*f))

---

[58] Of course, we could have introduced an analogous construction already on the level of the algebra of denotations. We didn't do so, since in our method, we want to sharply distinguish between the stages of building denotations and of building syntax. In our opinion a language designer should not think of syntax, when designing the core of the language represented by denotations.

then we decide not only about the functioning of a parser, but also about the meaning of the expression.

We shall not go into the technical details of a definition of RES assuming that it has been somehow (chosen and) defined. Let us think, therefore, about the definition of SEM, which is a composition of RES with the semantics of concrete syntax:

SEM = RES ● C2D

The definition of this semantics may be created algorithmically from the definition of C2D. Let's show it on the example of equation (7.5-2). Its colloquial counterpart will be the following:

```
SEMins.[ins] =
   RES.ins :: cre := cve                                 ➔
      assign.(C2Dcre.[cre], C2Dcve.[cve])
   RES.ins :: call cid-1.cid-2 ( val cap-1 ref cap-2)    ➔
      call-imp-pro.(C2Dcid.[cid-1] , C2Dcid.[cid-2] , C2Dcap.[cap-1] , A2Dcap.[cap-2])
   …
   RES.ins :: atin ; cin                                 ➔
      seq-ins.(C2Dcin.[atin], C2Dcin[cin])
```

In this definition we first restore a colloquial instruction ins into a corresponding concrete one RES.ins, and then we parse it to apply C2D cin. Note that C2Dcin also adds some parentheses (to instructions) by using C2A.

7.5.2 Why do we need a denotational semantics?

A denotational semantics of a programming languages constitutes a fundament for the realization of at least three goals:

1. to build an implementations of the language, i.e. an interpreter or compiler,
2. to write a concise, complete and consistent user manual,
3. to establish constructors of functionally correct programs.

Regarding the first goal, the definitional clauses of a denotational semantics may be regarded as procedures of an interpreter. They mutually call themselves, and call also constructors of denotation. As such they should be easily implementable. They also indicate a systematic way to the development of a compiler.

A denotational semantics is also an adequate starting point for writing a user manual. Even if a user is not prepared to read, and understand denotational equations, these equations constitute guidelines for an author of a manual. Translated into intuitive explanations — as we did in Sec. 6 — result with a manual that is consistent, complete and concise at the same time. An experiment of writing a manual in this way has been described in .

One of the well-known nightmares of manual reader is that manual usually don't keep up with the updates of implementations. The existence of a mathematical semantics of a language which should be conformant with both, the implementation and the manual, helps in keeping the adequacy of manuals.

It is also to be mentions in this place that although a denotational semantics needs not be a core of a manual, it should be contained in it as a standard to be referred to in cases of doubts. In such cases it may be practical to write semantic clauses in an unfolded form. E.g. instead or writing:

```
SEM.ins.[rex := vex] =
   assign.(SEM.rex.[rex], SEM.vex.[vex]
```

we may prefer to write

```
SEM.ins.[rex := vex].sta =
   is-error.sta                 ➔ error.sta
   let
      red   = SEM.rex.[rex]
```

```
   ved   = SEM.vex.[vex]
ved.sta = ?                   ➔ ?
ved.sta : Error               ➔ sta ◄ ved.sta
red.sta  : Error              ➔ sta ◄ red.sta
let
   val                               = ved.sta
   ref                               = red.sta
   (tok, (typ, yok, re-ota))         = ref
   (env, (obn, dep, st-ota, sft, 'OK'))  = sta
re-ota ≠ $ and re-ota ≠ st-ota ➔ 'reference not visible'
not ref VRA.cov val           ➔ 'incompatibility of types'
true                          ➔ (env, (obn, dep[ref/val], cov, st-ota, sft, 'OK'))
```

Whereas implementations and manuals may be created without a mathematical semantics — which, unfortunately, is a fairly common practice — it is hard to expect that we could create (mathematically) sound program construction rules without it. How to do when we have such semantics, we explain in Sec. 9.

8 SEMANTIC CORRECTNESS OF PROGRAMS

8.1 Historical remarks

Semantic correctness of programs, historically called *program correctness,* was a subject of investigations from the very beginning of the computer era. The earliest paper in this field — today practically forgotten — has been published by a British mathematician, Alan Turing[59], in 1949 [87]. Nearly twenty years later, in 1967, the same ideas were investigated again by an American scientist, Richard Floyd [53]. In 1978, the Association for Computing Machinery established the annual Turing Price "for outstanding achievements in informatics". One of the first winners of that price in 1978 was… Richard Floyd.

As far as we know, it has never been established if Floyd knew Turing's work. In the 1980-ties, A. Blikle wrote to Cambridge University on that issue. The only answer he received was substantial advice: do not try to build "yet another myth about Turing".

The work of Floyd introduced a fundamental concept of *an invariant of a program* and was dedicated to programs represented by graphical forms called *flow-diagrams* or *frow-charts.* In 1969, a British scientist, C.A.R Hoare (also a Turing Price winner), published a paper [61] concerning Floyd's ideas applied to *structural programs*, i.e., programs constructed with the help of sequential composition, if-then-else branching, and while loops. The works of C.A.R. Hoare and his followers, called *Hoare's logic*, were later summarised in two extensive monographs by K. Apt [4] and by K. Apt and H.R. Olderog [5].

The correctness of programs investigated by C.A.R. Hoare was later called *partial correctness*. A program is partially correct for a precondition prc and a postcondition poc if whenever prc is satisfied by an input state, and the execution of this program terminates, then the terminal state satisfies prc.

An alternative, or better a strengthening, of partial correctness is *total correctness,* introduced by E. W. Dijkstra in [50] and then investigated in detail in [51]. In this case, correctness means that the satisfaction of a precondition guarantees that the program terminates and satisfies the postcondition at the end.

Research devoted to program correctness was also developed in Poland. The first paper on that subject (although in an approach different from Hoare's) was published in 1971 by A. Mazurkiewicz [71]. A year later, during the first conference in a series of conferences on *Mathematical Foundations of Computer Science*[60], A. Blikle and A. Mazurkiewicz presented a joint paper [37] on program correctness based on an algebra of binary relations and covering recursive programs with nondeterminism. Nearly ten years later, A.Blikle published a paper [28] with a complete model of so-called *clean total correctness* for programs corresponding to arbitrary flow diagrams but without procedures. Blikle's correctness means that a correct program not only does not loop but also does not abort. This approach also gave rise to using three-valued predicates when talking about program correctness.

---

[59] Alan Turing (1912-1954) was one of the creators of the theory of computability. His model known today as *Turing machine* is regarded as one of fundamental concepts of this theory. Due to his work "On Computable Numbers, With an Application to the Entscheidungsproblem" Turing was considered as one of the greatest mathematicians of the world. Unfortunately he was also subject to a homophobic discrimination. When in 1952 police has learned about his homosexuality he was forced to choose between prison or hormonal therapy. He has chosen the latter but committed a suicide.

[60] This conference was organized in 1972 by a group of young researchers form the Institute of Computer Science of the Polish Academy of Sciences and the Department of Mathematics and Mechanics of Warsaw's University. Next year a similar conference was organized in Czechoslovakia witch gave rise to a long series of MFCS conferences. Since 1974 proceedings of these conferences were published by Springer Verlag in the series Lecture Notes in Computer Science.

In this place, we should also mention two fields of research developed at Warsaw University. The first was a formalized approach to program correctness based on algorithmic logic [10], where programs appear in logical formulas. The second [69] was much more engineering-oriented and split into three areas: *grammatical deduction, performance analysis of computing systems,* and *formal specification of software requirements*. An interesting application of the second approach is described in a paper by D.L. Parnas, G.J.K. Asmis, and J. Madey [79] devoted to software safety assessment for a Darlington Nuclear Power Generating Station (Canada) shutdown system.

Despite its undoubted scientific importance, the idea of proving programs correct was never widely applied in software engineering. In our opinion, this situation was due to the implicit assumption that programs come first and their proofs are built later. This order is natural in mathematics, where a theorem precedes its proof, but is somewhat unusual in engineering. Imagine an engineer who first constructs a bridge and only later performs all the necessary calculations. Such a bridge would probably collapse before its construction was completed, and in fact, this is what unavoidably happens with programs. The first version of a code usually does not work as expected. Consequently, a large part of the program-development budget is spent on testing and "debugging", i.e., on removing bugs introduced at the stage of writing the code. It is a well-known fact that all bugs can never be identified and removed by testing. Therefore, the remaining bugs are removed at the user's expense under the name of "maintenance". This process practically never terminates.

In this place, it is worth quoting a remark of Edsger W. Dijkstra that he called a "sad remark" [61]:

Since then we have witnessed the proliferation of baroque, ill-defined and, therefore, unstable software systems. Instead of working with a formal tool, which their task requires, many programmers now live in a limbo of folklore, in a vague and slippery world, in which they are never quite sure what the system will do to their programs. Under such regretful circumstances the whole notion of a correct program — let alone a program that has been proved to be correct — becomes void. What the proliferation of such systems has done to the morale of the computing community is more than we can describe.

Even though these words were written nearly half a century ago, and during this time, the reliability of hardware and the applicability of IT has increased by several orders of magnitude, the problems pointed out by E.W. Dijkstra are still there.

In this book (Sec. 8 and Sec. 9), we are trying to develop ideas sketched earlier by A. Blikle in [25] and [27], where instead of proving programs correct, a programmer develops programs using rules that guarantee program correctness. In such a framework, a software engineer works as an engineer who builds bridges, cars, or airplanes, and where products are created from correct components by using rules that guarantee the correctness of the result.

Since the rules for developing correct programs are derived from the rules of proving programs correct, we shall start with the latter. The discussion will be based on an algebra of binary relations since this leads to a relatively simple model where many technicalities of programming languages can be hidden. Of course, to apply these rules in a practical environment, they have to be expressed on the grounds of a mathematical model of a programming language. A language **Lingua-V** (V for "validation") with such a model is described in Sec. 9.

8.2 A relational model of nondeterministic programs

Each program, and each of its imperative components, defines an *input-output relation* (I-O relation) between its input states, and the corresponding output states. Of course, in a deterministic case, this relation is a function. Although programs in **Lingua** are deterministic, the discussion of a (possibly) non-deterministic case seems worthwhile, especially since it does not complicate the model.

---

[61] In [51] published in 1976 page 202.

Let S be an arbitrary, possibly infinite, set of elements called *states*. In **Lingua,** states are fairly complex items but in the abstract case, we do not need to assume anything about them. In the relational model programs are represented by binary relations over S, i.e., elements of the set:

Rel(S, S) = {R | R ⊆ S x S}

The fact that

a R b for a, b : S

means that there exists an execution of program R that starts in a and terminates in b.

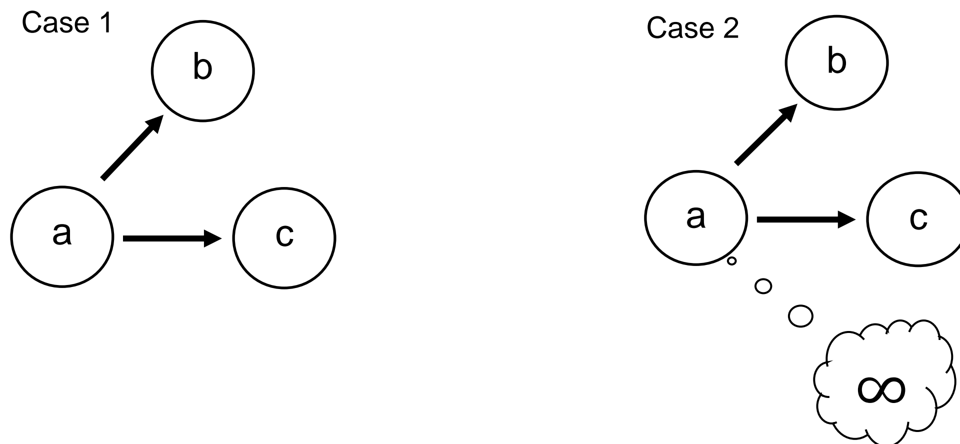


Fig. 8.2-1 Two nondeterministic cases

In a non-deterministic case, there may be more than one execution that starts in a. Some may terminate with another state, say c (Case 1 of Fig. 8.2-1), some others may be infinite (Case 2 of Fig. 8.2-1). In our model, the difference between Case 1 and Case 2 cannot be expressed. In both cases, we can only say that

a R b and a R c.

Note that due to the use of states which may carry errors, abortion of a computation from a to b means that b carries an error. This also means that if R is a function than the non-existence of a state b such that a R b means that a starts an infinite execution.

If we want to deal with infinite executions explicitly, we need a different concept of program denotations. Two such models were analyzed in [23]. One uses so-called δ-relations, where a R δ means that there exists an infinite computation that starts in a[62]. In this model, however, we cannot describe the fact that there are two or more different infinite computations that start from the same state. Such issues can be handled on the ground of the second model, where program denotations are sets of finite or infinite sequences of states called *bundles of computations*. Both approaches can be used in building denotational models of programming languages.

8.3 Iterative programs

In "prehistoric" informatics of the years 1940/1950, programs were written as lists of labeled instructions executed sequentially one after another unless a *jump instruction* goto interrupted that flow. With jump instructions one can build an arbitrary graph of elementary instructions called a *flow-diagram*. Early papers on program correctness were devoted to such programs later called *iterative programs*.

---

[62] In this model each δ-relation is a union of three set of pairs R ⊆ S x S, D ⊆ S x {δ} and {(δ, δ)}, where S and D may be empty.

A general relational model of an iterative program is the following fixed-point set of so called left-linear equations[63]:

$$X_1 = R_{11}\, X_1 \mid \dots \mid R_{1n}\, X_n \mid E_{1n}$$

$$\dots \tag{8.3-1}$$

$$X_n = R_{n1}\, X_1 \mid \dots \mid R_{nn}\, X_n \mid E_{nn}$$

that corresponds to a graph whose nodes are numbers $1,\dots,n$, each relation R_{ij} labelles a unique edge between i and j, and each E_{in} (exit relation) is a "dangling edge" that start on i, but does not point to any other node. The code of such a program may be written as an arbitrarily ordered[64] sequences of labelled instructions of the form:

i : **do** R_{ij} **goto** j and

i : **do** E_{in}.

If there is no instruction between i and j, then the relation R_{ij} is empty which means that there are no executions between i and j. Since the atomic instructions R_{ij} and E_{in} are not necessarily functions, such a program may have a non-deterministic character. For (8.3-1) to be deterministic, two conditions must be satisfied:

- all R_{ij} and E_{in} must be functions,
- for every i, all $R_{i1},\dots,R_{in}$ and E_{in} must have disjoint domains.

As has been proved in [28], if $(P_1,\dots,P_n)$ is the least solution of (8.3-1), then P_i is the input-output relation of the path from node i to node n. Therefore, if we assume that 1 represents the initial node, and n is the final node, then P_1 is the input-output relation (the denotation) of our program. The class of iterative programs understood in that way, together with their correctness-proof rules, were investigated in [23] and [28]. It is worth mentioning in this place that P_i's correspond to A. Mazurkiewicz *tail functions* [71] or D. Scott and Ch. Strachey *continuations* [84]. Both these models were published in 1971.

Programmers of the decade 1950/1960 were competing with each other in building more and more complicated flowchart programs that usually nobody except them was able to understand. Unfortunately, quite frequently, the authors themselves were not able to predict the behavior of such programs.

As a reaction to these problems, first algorithmic programming languages such as Fortran and Algol-60 were created. They were offering tools for *structural programming* such as sequential composition, if-then-else, and while[65]. Such programs were much easier to understand and also allowed for significant simplification of program-correctness proof rules.

In the sequel, we shall restrict our discussion to only three primary *structural constructors* since they allow for the implementation of any "implementable" function[66]:

1. sequential constructor denoted by a semicolon ";",
2. conditional constructor if-then-else-fi,
3. loop constructor while-do-od.

The sequential composition is the composition of relations (functions) as defined in Sec. 2.7. To define the remaining constructors, we have to introduce additional concepts. Since in our case the denotations of boolean expressions are three-valued partial functions, each of them will be represented by two disjoint set of states:

---

[63] They are called so because coefficients of variables X_i stand on their left-hand side. A symmetric model of right-linear equations of the general form $X = XR \mid Q$ has been analysed in [19].

[64] The execution of such a program does not depend on the order of its instructions since every instruction points to the instruction which should be executed as the next one.

[65] The author who introduced the term "structured programming" was a Dutch computer scientist Edsger Dijkstra (see [50] and [51]).

[66] Precisely speaking, any "computable" function. This claim has been known as Church's thesis. A formal proof of this thesis in shown in [18], and is based on a simple programming language with a (sort of) denotational semantics.

$C = \{s \mid p.s = tt\}$
$\neg C = \{s \mid p.s = ff\}$

Of course, if p is a two-valued total predicate, then C | ¬C = S, and therefore only one set is necessary to represent it. Notice also that our model does not distinguish between the two cases:

p.s : Error
p.s = ?

In both of them s : S – (C | ¬C). If we want to distinguish between these cased, we have to represent predicates by three disjoint sets:

$C = \{s \mid p.s = tt\}$
$\neg C = \{s \mid p.s = ff\}$
$eC = \{s \mid p.s : Error\}$

where S – (C | ¬C | eC) includes states initiating infinite executions of p. We are not going to do so, since in constructing correct programs we equally care about the avoidance of abortion and of infinite computations. Therefore we can identify these two cases in our model. Of course, in the denotational model of **Lingua** the abortion was distinguished from infinite looping, because the detection of te latter is not computable.

It may be interesting to see, how on the ground of our relational model, we can express the difference between McCarthy's and Kleene's operators of propositional calculus. E.g.

(A, ¬A) and-mc (B, ¬B) = (A ∩ B, ¬A | A ∩ ¬B) — McCarthy
(A, ¬A) and-kl (B, ¬B) = (A ∩ B, ¬A | ¬B) — Kleene

Now, let P and Q represent arbitrary programs and a pair of disjoint sets of states (C,¬C) — an arbitrary three-valued partial predicate. Our three structural constructors may be defined as particular cases of the universal set of equations (8.3-1). We recall (Sec. 2.7) that for any set of states A

[A] = {(a, a) | a : A}

is a subset of an identity relation (function). Now, the equational definitions of structural constructors are the following:

Sequential composition — P ; Q

X = P Y
Y = Q

Therefore by Theorem 2.4-2:

X = P Q

Conditional composition — **if** (C,¬C) **then** P **else** Q **fi**

X = [C] Y | [¬C] Z
Y = P
Z = Q

where [C] and [¬C] are identity functions. Therefore by Theorem 2.4-2::

X = [C] P | [¬C] Q

Loop — **while** (C,¬C) **do** P **od**

X = [C] P X | [¬C]

As is easy to prove in this case

X = ([C] P)* [¬C]

Summarizing our definitions:

P ; Q = P Q

if (C,¬C) **then** P **else** Q **fi** = [C] P | [¬C] Q
while (C,¬C) do P od = ([C] P)* [¬C]

At the end one methodological remark is necessary. Although in **Lingua** all programs are deterministic, hence correspond to functions rather than relations, in the relational theory of program correctness we shall mainly talk about arbitrary relations (with an exception of while loops), since in these cases nondeterminism does not lead to more complicated proof rules.

8.4 Procedures and recursion

The next step towards the development of structural-programming techniques was the introduction of procedures and, in particular — recursive procedures. On the ground of the algebra of relations mutually recursive procedures may be regarded as components of a vector of relations $(R_1,\ldots,R_n)$ which is the least solution of a set of fixed-point *polynomial equations* of the form:

$$X_1 = \Psi_1.(X_1,\ldots,X_n)$$

…

$$X_n = \Psi_n.(X_1,\ldots,X_n)$$

In these equations, each $\Psi_i(X_1,\ldots,X_n)$ is a polynomial, i.e., a combination of variables, say X, with constants, say A, B, C, by composition and union, e.g., AXYB | XXC. Such sets of equations may be regarded as single fixed-point equations in a CPO of relational vectors ordered component-wise, i.e., in the CPO over the carrier:

$$Rel(S,S)^{cn} = \{(R_1,\ldots,R_n) \mid R_i : Rel(S,S)\}$$

Every such set of polynomial equations defines a vectorial function:

$$\mathbf{\Psi} : Rel(S,S)^{cn} \mapsto Rel(S,S)^{cn}$$
$$\mathbf{\Psi}.(R_1,\ldots,R_n) = (\Psi_1.(R_1,\ldots,R_n),\ldots, \Psi_n.(R_1,\ldots,R_n))$$

If each Ψi is continuous in all its variables, then **Ψ** is continuous as well, and therefore Kleene's theorem holds (Sec. 2.4).

Since the correctness problem for recursive procedures is much more complicated than in the iterative case (see [5]), we shall investigate in Sec. 8.6.2 and Sec. 8.7.2 a simple scheme of a recursive procedure with only one procedural call that corresponds to an equation of the form:

$$X = HXT \mid E \qquad (8.4\text{-}2)$$

where H, T, E : Rel(S,S) are relations called the *head* the *tail* and the *exit* of the procedure, respectively. Although this is certainly not a general scheme for a recursive procedure, it is quite common in practice. This scheme will be referred to as a *simple recursion*.

Notice that (8.4-2) covers the case of the iterative instruction **while-do-od** with H = [C]P, T = [S] and E = [¬C].

8.5 Three concepts of program correctness

To express the property of program correctness on the ground of the algebra of binary relations, we shall use two operations of a composition of a relation with a set. Both are similar to sequential compositions of relations defined in Sec. 2.7. In the sequel A, B, C,… will denote subsets of the set of states S and P, Q, R,… will denote relations in Rel(S,S). Both operations are denoted by the same symbol "●", which has also been used for a composition of functions:

A●R = {s | (∃a:A) a R s} — *left composition*; the image of A by R
R●B = {s | (∃b:B) s R b} — *right composition*; the coimage of B by R.

In the sequel, the symbol of composition "●" will be omitted; hence we shall write AR and RA. Intuitively speaking (see Fig. 8.5-1):

- AR is the set of all final states of executions of R that start in A; notice however that some of them may be at the same time final states of executions that start outside A,
- RB is the set of all initial states of executions of R that terminate in B, but if R is not a function, then some of them may at the same time generate executions that terminate outside B or do not terminate at all.

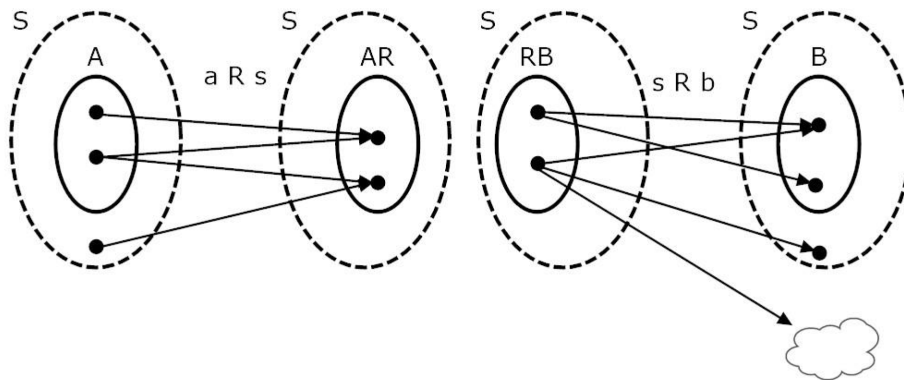


Fig. 8.5-1 Left- and right composition of a set with a relation

Both compositions of a relation with a set have properties similar to that of the composition of two relations. For instance, they are associative:

A(RQ) = (AR)Q
(RQ)B = R(QB)

and distributive over unions of sets and relations:

(A | B) R = (AR) | (BR)
A (R | Q) = (AR) | (AQ)
…

They are also monotone in each argument:

if A ⊆ B then AR ⊆ BR
if R ⊆ Q then AR ⊆ AQ

and analogously for right-hand-side composition. In fact, both operations are continuous in each argument. In the sequel, we shall assume that composition binds stronger than union hence we shall write

AR | BR instead of (AR) | (BR)

Lemma 8.5-1 *For any* A,B,C ⊆ S, *and* R : Rel(S,S) *the following equalities hold:*

1. [A]B = A∩B
2. A[B] = A∩B
3. (A∩B)R = A [B] R
4. R(A∩B) = R [A] B
5. (A∩B)R ⊆ C *is equivalent to* A[B]R ⊆ C
6. *if* A ⊆ [B]RC *then* (A∩B) ⊆ RC

Proofs are left to the reader.

Now we are ready to define three fundamental concepts concerning the correctness of programs: *partial correctness, weak total correctness,* and *clean total correctness*. All these concepts express the fact that if an input state of a program satisfies certain conditions, then the output state has expected properties. For

instance, we may expect that if a list-sorting program is given an appropriate list (precondition), then it will return a sorted list (postcondition).

With every property of states, we can unambiguously associate a set of states satisfying this property. As a consequence, the correctness of a program R for precondition A and postcondition B may be expressed in the algebra of relations and sets in the following way:

AR ⊆ B — *partial correctness* of R *for precondition* A *and postcondition* B;
(∀a:A) *if* (∃ b) aRb, *then* b:B

A ⊆ RB — *weak total correctness*[67] of R *for precondition* A *and postcondition* B;
(∀a:A) (∃ b) aRb *and* b:B

Partial correctness means that every execution that starts in A, if it terminates, then it terminates in B. Set A is called *partial precondition,* and B is called *partial postcondition*. If B does not contain error-carrying states then we talk about *clean partial correctness*.

Weak total correctness means that for every state a : A, there exists an execution that starts in a and terminates in B. Set A is called *weak total precondition,* and B is called *weak total postcondition*. The adjective "weak" expresses the fact that the existence of an execution from a to B does not exclude that other executions starting with a may terminate outside B or do not terminate at all. Similarly as in the former case, if B does not contain error-carrying states then we talk about *weak clean total correctness*.

Both defined concepts of program correctness were historically introduced for deterministic programs, i.e., for the case where R was a function. In such cases, the inclusion A ⊆ RB means that each execution of R that starts in A terminates in B. That property will be called *clean total correctness* and programs with this property will be said to be *totally correct with clean termination*. Our validating language described in Sec. 9 will include program-construction rules that guarantee clean total correctness of constructed programs.

As is easy to see, in the non-deterministic case, none of the partial and total correctness is stronger than the other. Indeed, partial correctness does not imply termination, and the existence of one terminating execution from a to B does not mean that any terminating execution starting in a will terminate in B.

In the deterministic case, however, total correctness obviously implies partial correctness. i.e., for any partial function F : S → S,

A ⊆ FB implies AF ⊆ B (8.5-1)

The following implication is also true:

if AF ⊆ B and for every a : A, F.a is defined then A ⊆ FB (8.5-2)

Both observations lead to the following theorem:

Theorem 8.5-1 *If* F *is a function, then for any* A,B ⊆ S *the following facts are equivalent:*

- A ⊆ FB — *total correctness of* F *wrt* A *and B*
- AF ⊆ B *and* A ⊆ FS — *partial correctness of* F *wrt* A *and* B, *plus termination of* F *on* A ■

Clean termination of a deterministic program F on A means that F is a total function on A, and F.a never carries an error.

We say that a deterministic program has a *halting property in* A, if no execution of that program that starts in A is infinite.

For many "practical programs", the halting property may be so obvious that it does not need a formal proof. For instance, the program:

---

[67] In the earlier versions of the book the weak total correctness of relations was called just total correctness. Krzysztof Apt convinced us that such wording may lead to misunderstanding. He also pointed out that in [6] written by him with two other authors the notion of weak total correctness is used in a slightly different way. It is used in the context of distributed programs and combines partial correctness with absence of failures and divergence freedom.

```
pre n, m > 0
   x := 1; y := m;
   while x < n
   do;
      x := x+1; y := y*m
   od
post y = m^n
```

obviously halts for every n. However, there are cases where the halting property may be far from evident, even for very simple programs. One such program is displayed on the front of Warsaw University Library:

```
x := n;
while x > 1
   do
      if x mod 2 = 0 then x := x/2 else x := 3x + 1 fi
   od
```

Under this program we see the following question: "Why for every n > 0 this program stops?". This question is, however, not adequate, since today we do not know, if this program has a halting property. It expresses a well-known *Collatz hypothesis* formulated in 1937 and not answered till today. At the date, we are writing these words (February 2024), it was only proved[68] that the hypothesis is true for all $n < 5*2^{69}$.

A similar situation concerns *Fermat's theorem*[70] that was announced in the year 1637 and proved only in 1994 by a British mathematician Andrew Wiles. His proof is 100 pages long and uses an advanced topological theory of elliptic curves. Fermat theorem can be also formulated as a halting problem.

On the ground of the theory of computability, it has been proved by Alan Turing that there is no algorithm which given a program[71] and an input state could check in a finite time, if this program terminates for this input state.

Theorem 8.5-2 *In the general case, the termination property of programs is not decidable.* ■

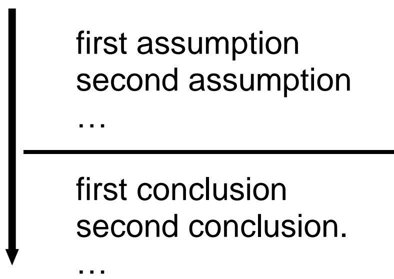
In the sequel, proof rules for program correctness will be expressed by showing in which way the correctness of composed programs may be proved by proving the correctness of their components. These rules will be written in the following form:

[68] One could (naïvely) expect that this result was proved by a simple checking in utilizing an ultra-fast computer. However, as is easy to calculate, if we assume that the execution of Collatz program for any $n < 5*2^{86}$ takes on the average 1 nanosecond, then such a check would take a time longer than 10^{65} times the age of the universe.

[69] Andrzej Blikle once fell victim to this hypothesis, when he was reporting his work on total correctness of programs at the University of Saarbrücken. When he said that with his method one can easily prove the termination of a program, a listener asked him to illustrate this fact on a simple example, and gave him the Collatz program. Blikle did not know this example, so he wrote the program on the board and proceeded to analyze it. Since he was not able to solve the problem off hand, he said: "I will think about this problem in the evening". But in the evening he still did not have a proof. What a shame — such a simple program, and he cannot cope with it. After returning to Warsaw he showed the problem to his colleagues, and was enlightened that he was not the only one who was not able to prove the Collatz's hypotheses.

[70] This theorem claims that for no integer n > 2 there exist three positive integers x, y, z that satisfy the equality $x^n + y^n = z^n$. That theorem had been written in 1637 by Pierre de Fermat on the margin of a book together with a commentary that he found a "marvellously simple proof" of the theorem which was however too long to fit to the margin. The theorem has been proved by Andrew Wiles in 1993, and his proof was more than 100 pages long.

[71] In the original work of A. Turing programs were represented by Turing machines, but since then in became a known fact that for every program there is a (functionally) equivalent Turing machine, and vice versa (e.g. cf. [64]).



where the arrow shows the direction of implication. In some rules, we have both-sided arrows, which means that the implication is of the **iff**-type. The list of assumptions and of conclusions are understood as corresponding conjunction.

It should be emphasized in this place that in our approach to program correctness we are not building any "logic of programs" in Hoare's or Dijkstra's style. We only construct a set-theoretical model of programs where the latter are represented by binary relations (or functions). On the ground of this model, program correctness is expressed by inclusions of the form $AP \subseteq B$ or $AP \subseteq B$. Then, we formulate and prove some lemmas which may be used either in proving programs correctness, or in building correct programs. In short, these lemmas will be called *proof rules* or *construction rules* depending on the way we shall use them.

In the end, one comment about using single sets of states A or B, rather than pairs $(C,\neg C)$, to represent three-valued pre- and post-conditions. In fact in using pre- and post-conditions, we are interested only in their "domains of satisfaction", i.e., in the first elements of each pair $(C,\neg C)$. For instance, in proving the correctness of a program with a precondition:

$$1/x > 2 \qquad (*)$$

we are only interested in the behavior of the program whenever our precondition is satisfied. We do not care about that behavior in all other cases. If, however, condition (*) would be used as a boolean expression of an **if-then-else-fi** instruction, then it must be represented by a pair of sets (cf. Rule 8.6.1-2)

8.6 Partial correctness

Although our primary concern is total correctness of programs, the methods of proving partial correctness are of interest too since in the deterministic case, proof of total correctness may be reduced to a proof of partial correctness plus a proof of termination (cf. (8.5-2)). In turn, although in the general case termination property is not decidable, in many practical cases it may be quite easy to prove.

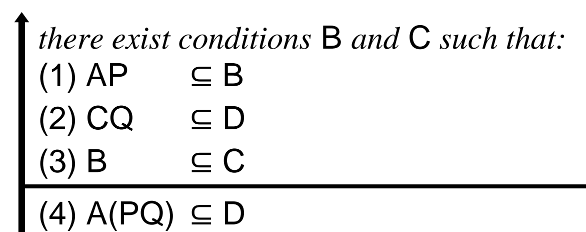
8.6.1 Sequential composition and branching

When defining program correctness proof rules, it is worth distinguishing between two classes of program constructors: *simple constructors* that do not introduce repetition mechanisms and *recursive constructors* that introduce such mechanisms. The former are defined by composition and union of relations; the latter require fixed-point equations. From this perspective, iteration is a particular case of recursion.

The most frequently used simple constructors of programs are sequential composition and branching.

Rule 8.6.1-1 Partial correctness of a sequential composition

For arbitrary $A,D \subseteq S$ *and* $P,Q : Rel(S,S)$ *the following rule is satisfied:*



Proof From (1), (2) and the monotonicity of composition

$(AP)Q \subseteq CQ \subseteq D$

hence from the associativity of composition

$A(PQ) \subseteq D.$

To prove the bottom-to-top implication is sufficient to set

$B = C = AP$

Hence $AP \subseteq B$ and $BQ = APQ \subseteq D$ ■

Rule 8.6.1-2 Partial correctness of if-then-else-fi

For arbitrary $A,D,C,\neg C \subseteq S$ *and* $P,Q : Rel(S,S)$, *if* $C \cap \neg C = \varnothing$, *then the following rule is satisfied:*

(1) $(A \cap C)P \subseteq B$
(2) $(A \cap \neg C)Q \subseteq B$
---
(3) A **if** (C, ¬C) **then** P **else** Q **fi** ⊆ B

The proof is obvious.

In the end, three more rules which follow directly from the monotonicity of composition of a set with a relation.

Rule 8.6.1-3 Strengthening a partial precondition

For every $P : Rel(S,S)$ *and any* $A,B,C \subseteq S$ *the following rule holds:*

$AP \subseteq B$
$C \subseteq A$

$CP \subseteq B$

Rule 8.6.1-4 Weakening a partial postcondition

For every $P : Rel(S,S)$ *and any* $A,B,C \subseteq S$ *the following rule holds:*

$AP \subseteq B$
$B \subseteq C$

$AP \subseteq C$

Rule 8.6.1-5 The conjunction and disjunction of pre- and postconditions

For every $P : Rel(S,S)$ *and any* $A,B,C,D \subseteq S$ *the following rule holds:*

$AP \subseteq B$
$CP \subseteq D$

$(A \cap C)P \subseteq B \cap D$
$(A \mid C)P \subseteq B \mid D$

In the present section we skip the problem of proving properties of atomic components of programs such as, e.g., assignments or variable declarations since they are not expressible in the model of abstract binary. This issue will be discussed in Sec. 9 where **Lingua-V** enters the game.

8.6.2 Recursion and iteration

In order to formulate proof rules for mutually recursive procedures, we generalize the operation of composition of relations with relations and with sets to the case of vectors of respectively relations and sets:

$(P_1,\ldots,P_n)\ (R_1,\ldots,R_n) = (P_1R_1,\ldots,P_nR_n)$

and analogously for the composition of a relation with sets. In an obvious way, we can also generalize the inclusion of sets to the inclusion of vectors:

$(A_1,\ldots,A_n) \subseteq (B_1,\ldots,B_n)$ means $A_1 \subseteq B_1$ and … and $A_n \subseteq B_n$

For simplicity, the inclusion between vectors of sets is denoted by the same symbol as the inclusion of sets. In the sequel, vectors of sets and relations as well as operations on them will be written with boldface characters.

A vector of relations $\mathbf{R}$ is said to be *partially correct* wrt the vectors of sets $\mathbf{A}$ and $\mathbf{B}$ (with appropriate numbers of elements) iff $\mathbf{A}\ \mathbf{R} \subseteq \mathbf{B}$. The notion of a continuous function is generalized to the case of vectorial functions in an obvious way.

Now we can formulate partial-correctness proof rule in the general case of fixed-points of continuous functions on vectors of relations.

Rule 8.6.2-1 Partial correctness of a vector of relations defined by a fixed-point equation

For every continuous function $\mathbf{\Psi} : \mathrm{Rel(S,S)}^{cn} \mapsto \mathrm{Rel(S,S)}^{cn}$, *if* $\mathbf{R}$ *is the least solution of the equation* $X = \mathbf{\Psi}.X$, *then for any* $\mathbf{A},\mathbf{B} : S^{cn}$ *the following rule holds, where* $\mathbf{Ø} = (Ø,\ldots, Ø)$ *is a* n*-element vector of empty relations:*

there exists a family of (vectors of) preconditions $\{\mathbf{A}_i \mid i \geq 0\}$
and a family of (vectors of) postconditions $\{\mathbf{B}_i \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ \mathbf{A} \subseteq \mathbf{A}_i$
(2) $(\forall i \geq 0)\ \mathbf{A}_i\ \mathbf{\Psi}^i.\mathbf{Ø} \subseteq \mathbf{B}_i$
(3) $\mathrm{U}\{\mathbf{B}_i \mid i \geq 0\} \subseteq \mathbf{B}$
___
(4) $\mathbf{AR} \subseteq \mathbf{B}$

Proof Form Kleene's theorem (Sec. 2.4)

$\mathbf{R} = \mathbf{U}\ \{\mathbf{\Psi}^i.\mathbf{Ø} \mid i \geq 0\}$

Adding the components of (1) sidewise we obtain

$\mathbf{U}\ (\mathbf{A}_i\ \{\mathbf{\Psi}^i.\mathbf{Ø} \mid i \geq 0\} \subseteq \mathbf{U}\{\mathbf{B}_i \mid i \geq 0\}$

hence from (1) and (3), we have (4). To prove the bottom-up implication, we assume

$\mathbf{B}_i = \mathbf{A}\ (\mathbf{\Psi}^i.\mathbf{Ø})$ for $i \geq 0$ and

$\mathbf{A}_i = \mathbf{A}$ ■

From this rule, we obtain immediately a rule for single recursion, i.e., where $n = 1$:

Rule 8.6.2-2 Partial correctness of a relation defined by a fixed-point equation

For every continuous function $\Psi : \mathrm{Rel(S,S)} \mapsto \mathrm{Rel(S,S)}$, *if* R *is the least solution of the equation* $X = \Psi.X$, *then for any* $A,B \subseteq S$ *the following rule holds:*

there exists a family of preconditions $\{Ai \mid i \geq 0\}$
and a family of postconditions $\{Bi \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ A_i\ \Psi_i.Ø \subseteq B_i$
(2) $(\forall i \geq 0)\ A \subseteq A_i$
(2) $\mathrm{U}\{B_i \mid i \geq 0\} \subseteq B$
___
(3) $AR \subseteq B$

We can also formulate more specific rules for each particular polynomial function, e.g., for the simple-recursion constructor as defined in Sec. 8.4. Below two versions of such a rule:

Rule 8.6.2-3 Partial correctness of a relation defined by simple recursion (version 1)

For any H,T,E : Rel(S,S), *if the relation* R *is the least solution of the equation*

X = HXT | E

then for any A,B ⊆ S *the following rule holds:*

there exists a family of preconditions $\{A_i \mid i \geq 0\}$
and a family of postconditions $\{B_i \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ A_i H_i E T_i \subseteq B_i$
(2) $(\forall i \geq 0)\ A \subseteq A_i$
(2) $\cup\{B_i \mid i \geq 0\} \subseteq B$

---

(3) AR ⊆ B

The proof follows immediately from Rule 8.6.2-2 and from the fact that, as is easy to prove,

$R = \cup\{H^i E T^i \mid i \geq 0\}$ ■

The following top-down-implication rule with a stronger assumption may be useful as well:

Rule 8.6.2-4 Partial correctness of a relation defined by simple recursion (version 2)

For any H,T,E : Rel(S,S), *if the relation* R *is the least solution of the equation*

X = HXT | E

then for any A,B ⊆ S *the following rule holds:*

(1) (∀ Q) (AQ ⊆ B *implies* A(HQT) ⊆ B)
(2) AE ⊆ B

---

(3) AR ⊆ B

Proof From (1) and (2) we can prove by induction that for every $i \geq 0$:

$A (H^i E T^i) \subseteq B$

and, therefore, by side-wise summation, we get (3). ■

Rule 8.6.2-5 A Partial correctness of a relation defined by simple recursion (version 3)

For any H,T,E : Rel(S,S), *if the relation* R *is the least solution of the equation*

X = HXT | E

then for any A,B ⊆ S *the following rule holds:*

(1) AH ⊆ A
(2) AE ⊆ B
(3) BT ⊆ B

---

(4) AR ⊆ B

Proof The three inclusions (1), (2), and (3) imply that for any $i > 0$, we have

$A (H^i E T^i) \subseteq A E T^i \subseteq B T^i \subseteq B$. ■

Now let us denote by

while (C, ¬C) **do** P **od**

the least solution of the equation

$X = [C]PX \mid [\neg C]$.

Setting H = [C]P, T = [S] and E = [¬C] from both general rules we can draw rules for while-do-od iteration:

Rule 8.6.2-6 Partial correctness of while-do-od loop (version 1)

For every relation P : Rel(S,S), *any disjoint* C, ¬C ⊆ S, *and any* A,B ⊆ S *the following rule holds:*

there exists a family of postconditions $\{B_i \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ A ([C]P)^i [\neg C] \subseteq B_i$
(2) $\cup\{B_i \mid i \geq 0\} \subseteq B$
---
(3) A **while** (C, ¬C) **do** P **od** ⊆ B

Rule 8.6.2-7 Partial correctness of while-do-od loop (version 2)

For every relation P : Rel(S,S), *any disjoint* C, ¬C ⊆ S, *and any* A, B ⊆ S *the following rule holds:*

(1) (∀ Q) AQ ⊆ B *implies* A [C]QP ⊆ B
(2) A[¬C] ⊆ B
---
(3) A **while** (C, ¬C) **do** P **od** ⊆ B

■

In the literature, the following rule is also well known, although it is usually formulated for the case of two-valued predicates, i.e. where C | ¬C = S

Rule 8.6.2-8 Partial correctness of while-do-od loop (version 3)

For every relation P : Rel(S,S), *for any disjoint* C,¬C ⊆ S, *any* A, B ⊆ S, *the following rule is satisfied:*

there exists N ⊆ S *(called loop invariant) such that*:
(1) (N ∩ C) P ⊆ N
(2) A ⊆ N
(3) N [¬C] ⊆ B
---
(4) A **while** (C, ¬C) **do** P **od** ⊆ B

■

Proof Let (1) – (3) be satisfied. Since

$(N \cap C)\ P = N\ [C]\ P$

from (1) we can prove by induction:

$N([C]P)^i \subseteq N$ for all $i \geq 0$

Therefore and from (2)

$A([C]P)^i \subseteq N$ for all $i \geq 0$

hence from (3)

$A([C]P)^i[\neg C] \subseteq N[\neg C] \subseteq B$ for all $i \geq 0$

In summing these inclusions sidewise, we get (4). Now assume that (4) is satisfied and let us set:

(5) $N = A([C]P)^*$

Therefore and from (4) we get $N[\neg C] \subseteq B$, hence (3). In turn (5) is equivalent to

$N = A \mid A([C]P)^+$,

hence (2). To prove (1) notice that:

$(N \cap C)P = N[C]P = A[C]P \mid A([C]P)^+[C]P = A([C]P)^+ \subseteq N$ ■

8.7 Weak total correctness

Rules for weak total correctness are used to prove that if an input state of a program satisfies a precondition, then at least one execution of that program will terminate with postconditions satisfied. If a program is deterministic, then weak total correctness coincides with clean total correctness which means that the unique execution of a program terminates with a state satisfying a postcondition.

8.7.1 Sequential composition and branching

Rule 8.7.1-1 Weak total correctness of a composition

For any $A,D \subseteq S$ *and* $P,Q : Rel(S,S)$ *the following rule holds:*

there exist conditions B *and* C *such that*
(1) $A \subseteq PB$
(2) $C \subseteq QD$
(3) $B \subseteq C$

---

(4) $A \subseteq (PQ)D$

Proof. From (1), (2) and (3) we immediately have:

$A \subseteq PB \subseteq PC \subseteq P(QD) = (PQ) D$.

Now assume that $A \subseteq (PQ)D$, which means that $A \subseteq P(QD)$. Assuming $B = C = QD$ we get (1) and (2). ■

Rule 8.7.1-2 Weak total correctness of if-then-else[72]

For any $A,B,C,\neg C \subseteq S$ *and* $P,Q : Rel(S,S)$, *if* $C \cap \neg C = \varnothing$, *then the following rule is satisfied:*

(1) $A \cap C \subseteq PB$
(2) $A \cap \neg C \subseteq QB$
(3) $A \subseteq C \mid \neg C$

---

(4) $A \subseteq$ **if** $(C, \neg C)$ **then** P **else** Q **fi** B

Proof. Let (1) – (3) be satisfied. Then:

$[C] (A \cap C) \subseteq [C] PB$
$[\neg C] (A \cap \neg C) \subseteq [\neg C] QB$

Adding the inclusions sidewise:

$[C] (A \cap C) \mid [\neg C] (A \cap \neg C) \subseteq [C] PB \mid [\neg C] QB = ([C]P \mid \mid [\neg C] Q) B$

The following equalities are also true

$[C] (A \cap C) = A \cap C$

[72] Notice that in the case of two-valued predicates, condition (3) would not be necessary, since in that case $C \mid \neg C = S$.

and analogously for $\neg C$. Hence and from (3)

$$[C]\ (A \cap C)\ \mid [\neg C]\ (A \cap \neg C) = (A \cap C)\ \mid (A \cap \neg C) = A$$

and finally

$$(4)\ A\ \subseteq [C]\ PB \mid [\neg C]\ QB$$

In turn, (4) implies $A \subseteq C \mid \neg C$, and from (4) and the fact that C and $\neg C$ are disjoint, follow (1) and (2). ■

Observe the assumption (3) in our rule. In the case of classical predicates where $C \mid \neg C = S$, this condition is a tautology.

In the end, three more rules for pre- and postconditions analogous to the respective rules for partial correctness.

Rule 8.7.1-3 The strengthening of a weak total precondition

For every P : Rel(S,S) *and any* $A,B,C \subseteq S$ *the following rule holds:*

$A \subseteq PB$
$C \subseteq A$
―――――――
$C \subseteq PB$

Rule 8.7.1-4 The weakening of a weak total postcondition

For every P : Rel(S,S) *and any* $A,B,C \subseteq S$ *the following rule holds:*

$A \subseteq PB$
$B \subseteq C$
―――――――
$A \subseteq PC$

Rule 8.7.1-5 The conjunction and disjunction of conditions

For every P : Rel(S,S) *and any* $A,B,C,D \subseteq S$ *the following rule holds:*

$A \subseteq PB$
$C \subseteq PD$
―――――――
$A \cap C \subseteq P(B \cap D)$
$A \mid C \subseteq P(B \mid D)$

The proofs of the last three rules follow directly from the definitions of total correctness. Our last rule in this section concerns resilient conditions.

Rule 8.7.1-6 Propagation of resilient conditions

For every P : Rel(S,S) *and any* $A,B,C \subseteq S$ *the following rule holds:*

(1) $A\ \ \subseteq PB$
(2) $CP \subseteq C$
―――――――
$A \cap C \subseteq P(B \cap C)$

In this rule, C is said to be *resilient to* P, because its satisfaction is not violated by P. This rule, although quite simple, has a practical value, since it will be applied in all situations where a certain property of a state once established, remains in force till the end of the execution of a program. E.g., once we declare a variable

it remains declared during the whole (remaining) lifetime of the hosting program. The proof of this rule is the following:

From (1) by Rule 8.7.1-3 we have $A \cap C \subseteq PB$. Consequently, if $a : A \cap C$ then there exists a state b such that $a\ P\ b$ and $b : B$. At the same time, since $a : C$, then by (2) $b : C$, hence $b : B \cap C$. ■

8.7.2 Recursion and iteration

Similarly, as in the case of partial correctness, we start from the case of a general recursive operator.

Rule 8.7.2-1 Weak total correctness of a vector defined by a general fixed-point equation

For every continuous function $\mathbf{\Psi} : \mathbf{Rel(S,S)}^{cn} \mapsto \mathbf{Rel(S,S)}^{cn}$, *if* $\mathbf{R}$ *is the least solution of* $\mathbf{X} = \mathbf{\Psi.X}$, *then the following rule holds, where* $\mathbf{Ø} = (Ø,\dots,Ø)$:

there exists a family of preconditions $\{\mathbf{A}_i \mid i \geq 0\}$
and a family of postconditions $\{\mathbf{B}_i \mid i \geq 0\}$ *such that*
(1) $(\forall\ i \geq 0)\ \mathbf{A}_i \subseteq (\mathbf{\Psi}^i.\mathbf{Ø})\mathbf{B}_i$
(2) $\mathbf{A} \subseteq \cup\{\mathbf{A}_i \mid i \geq 0\}$
(3) $(\forall\ i \geq 0)\ \mathbf{B}_i \subseteq \mathbf{B}$

---

(4) $\mathbf{A} \subseteq \mathbf{RB}$

Proof If $\mathbf{R}$ is the least fixed point of $\mathbf{\Psi}$, then from the continuity of $\mathbf{\Psi}$

(4) $\mathbf{R} = \mathbf{\cup}\{\mathbf{\Psi}^i.\mathbf{Ø} \mid i \geq 0\}$

Adding sidewise inclusions (1) we have

$\mathbf{\cup}\ \{\mathbf{A}_i \mid i \geq 0\} \subseteq \mathbf{\cup}\ (\{\mathbf{\Psi}^i.\mathbf{Ø} \mid i \geq 0\}\ \mathbf{B}_i)$

Hence from (2) and (3), we have (4). Now assume that $\mathbf{A} \subseteq \mathbf{RB}$ which means that

$\mathbf{A} \subseteq \mathbf{\cup}\{\mathbf{\Psi}^i.Ø \mid i \geq 0\}\ \mathbf{B}$

Let for $i \geq 0$

$\mathbf{A}_i = (\mathbf{\Psi}^i.Ø)\ \mathbf{B}$ and

$\mathbf{B}_i = \mathbf{B}$

Then obviously (1), (2), and (3) are satisfied. ■

From this rule for $n = 1$, we immediately conclude the next rule

Rule 8.7.2-2 Weak total correctness of a relation defined by a general fixed-point equation

For every continuous function $\Psi : Rel(S,S) \mapsto Rel(S,S)$, *if* R *is the least solution of an equation* $X = \Psi.X$, *then the following rule holds:*

there exists a family of preconditions $\{A_i \mid i \geq 0\}$
and a family of postconditions $\{B_i \mid i \geq 0\}$ *such that*
(1) $(\forall\ i \geq 0)\ A_i \subseteq (\Psi^i.Ø)B_i$
(2) $A \subseteq \cup\ \{A_i \mid i \geq 0\}$
(3) $(\forall\ i \geq 0)\ B_i \subseteq B$

---

(4) $A \subseteq RB$

Rule 8.7.2-3 Weak total correctness of a relation defined by simple recursion (version 1)

If relation R *is the least solution of the equation* X = H X T | E *then the following rule holds:*

there exists a family of preconditions $\{A_i \mid i \geq 0\}$
and a family of postconditions $\{B_i \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ A_i \subseteq (H^i\ E\ T^i)\ B_i$
(2) $A \subseteq U\ \{A_i \mid i \geq 0\}$
(3) $(\forall i \geq 0)\ B_i \subseteq B$

---

(4) $A \subseteq RB$

Proof Define

$\Psi.X = H\ X\ T \mid E$

In this case

$\Psi^0.\varnothing = E$

$\Psi^1.\varnothing = \Psi.(\Psi^0.\varnothing) = H\ (\Psi^0.\varnothing)\ T \mid E = H\ E\ T \mid E$

$\Psi^2.\varnothing = \Psi.(\Psi^1.\varnothing) = H\ (\Psi^1.\varnothing)\ T \mid E = H\ (\Psi^1.\varnothing)\ T \mid E = H^2\ E\ T^2 \mid H^1\ E\ T^1 \mid E$

Therefore, by induction, for any $n \geq 0$

$\Psi^i.\varnothing = U\ \{ H^i\ E\ T^i \mid i=1,2,\ldots n\} \mid E = = U\ \{ H^i\ E\ T^i \mid i=0,1,\ldots n\}$

Now, by (1) and the monotonicity of composition of a relation with a set, we have for every $i \geq 0$

$A_i \subseteq H^i\ E\ T^i\ B_i \subseteq (U\ \{H^i\ E\ T^i \mid 1=0,\ldots,n\}\ )\ B_i \subseteq (\Psi^i.\varnothing)\ B_i$

From this inclusion together with (2), (3) and Rule 0-2, we conclude

$A \subseteq RB$

In turn, if the inclusion is satisfied, then we set

$A_i = (\Psi^i.\varnothing)\ B$

$B_i = B$

With this settings (1) and (3) are obviously satisfied, and (2) is satisfied because

$A \subseteq RB \subseteq U\{\ \Psi^i.\varnothing \mid i\geq 0\}\ B = U\{\ (\Psi^i.\varnothing)\ B \mid i\geq 0\} = U\ \{A_i \mid i \geq 0\}$ ■

Rule 8.7.2-4 Clean total correctness of a function defined by simple recursion (version 2)

If F *is the least solution of the equation* X = HXT | E *where* H, T, *and* E *are functions and the domains of* H *and* E *are disjoint, then the following rule holds:*

(1) $(\forall Q)$ ($AQ \subseteq B$ *implies* $A(HQT) \subseteq B$)
(2) $AE \subseteq B$
(3) $A \subseteq FS$

---

(3) $A \subseteq FB$

Proof As is easy to prove, for any H, T, and E the least solution of our equation is

$U\{\ H^n\ E\ T^n \mid n\geq 0\ \}$

and if additionally H, T, and E are functions and the domains of H and E are disjoint, then this solution is a function. Now, by (1), (2) and the Rule 8.6.2-4, AF ⊆ B, i.e., F is partially correct wrt A and B. Since (3) means that F is total on A, by Theorem 8.5-1 we can claim that it is totally correct wrt A and B. ■

From Rule 8.7.2-3 we can immediately derive our first rule about while-do-od instruction based on the observation that **while** (C, ¬C) **do** P **od** is the least solution of the equation

X = [C]PX | [¬C].

Let then R be the least solution of this equation, i.e.,

R = ([C]P)*[¬C].

Rule 8.7.2-5 Clean total correctness for nondeterministic while-do-od

there exists a family of preconditions $\{A_i \mid i \geq 0\}$
and a family of postconditions $\{B_i \mid i \geq 0\}$ *such that*
(1) $(\forall i \geq 0)\ A_i \subseteq ([C]P)^i[\neg C]\ B_i$
(2) $A \subseteq \cup \{A_i \mid i \geq 0\}$
(3) $(\forall i \geq 0)\ B_i \subseteq B$

---

(4) A ⊆ RB

The most commonly known version of a rule for **while-do-od** concerns a deterministic case, and does not require the construction of two infinite families of conditions. It is also based on a well-known method of proving the halting property of a loop. First, we introduce two auxiliary concepts.

We say that a function F : S → S has *limited replicability property* in a set N ⊆ S, if there exists no infinite sequence of the form: s, F.s, F.(F.s),… in N.

A partially ordered set (U, >) is said to be *well-founded*, if there is no infinite decreasing sequence in it, i.e., a sequence $u_1 < u_2 <$ … The following obvious lemma is useful in proving the limited replicability of a function F : S → S.

Lemma 8.7.2-1 If there exists a well-founded set (U, <) and a function K : N ↦ U such that for any a : N, F.a = !, F.a : N and

K.a > K.(F.b)

then F has limited replicability in N. ■

Now we can formulate our rule.

Rule 8.7.2-6 Clean total correctness of a deterministic while-do-od loop

For any function F : S → S, *any* A,B,N ⊆ S, *and any disjoint* C,¬C ⊆ S

(1) A ⊆ N
(2) N ⊆ C | ¬C
(3) N ∩ ¬C ⊆ B
(4) N ∩ C ⊆ FN (clean total correctness of F)
(5) [C]F *has limited replicability in* N

---

(6) A ⊆ **while** (C,¬C) **do** F **od** B

Proof Assume that (1), (2), (3) are satisfied but the inclusion

N ⊆ ([C]F)*[¬C]S.

does not hold. In that case, there exists s_0 : N, that does not belong to

$$([C]F)^*[\neg C]S = ([C]F)^+[\neg C]S \mid \neg C,$$

and therefore s_0 does not belong to ¬C. From there, by (3), s_0 : N∩C, and therefore by (4), there exists s_1 such that [C]F.s_0 = s_1 and s_1 : N. Therefore by (3)

$$s_1 : C \mid \neg C.$$

Now, s_1 cannot belong to ¬C, since then s_0 would belong to

$$[C]F[\neg C]S$$

which is a subset of $([C]F)^*[\neg C]S$. Reasoning in this way, we could prove by induction that for any $n \geq 0$ there exists a sequence s_i : i = 0,1,…n such that s_0 : N and

$$s_i \ [C]F \ s_{i+1} \text{ and } s_i : N \text{ for } i = 0,1,\ldots,n$$

Since F is a function, this implies the existence of an infinite sequence

$$s_i \ [C]F \ s_{i+1} \text{ and } s_i : N \text{ for } i = 0,1,\ldots$$

which contradicts (5). ■

9 VALIDATING PROGRAMMING

Generally speaking, by *validating programming,* we shall mean such program creation techniques that ensure clean total-correctness of programs wrt their specifications. In our approach, programs and their specifications will constitute syntactic components of *metaprograms*. This technique was already announced in Sec. 1.1 and its abstract mathematical foundations were described in Sec. 8. The present section is devoted to the techniques of developing *correct metaprograms* written in an extended programming language, **Lingua-V,** which includes **Lingua**.

An approach that gave rise to validated programming was proposed by A.Blikle in papers [25], [26] and [27] published at the turn of the decades 1970s and 1980s. In writing these papers, he concluded that to create a language with rules that guarantee program correctness, one has to equip this language with mathematical semantics. This observation provoked his further research described in [30], [32] and [33] which is now continued in our book.

9.1 Languages of validating programming

From a pure logical perspective metaprograms may be seen as theorems that claim the correctness of programs that they (syntactically) include. An example of such a metaprogram written in **Lingua-V,** may be the following:

```
pre x,k is integer and k > 0:
    x := 0;
    asr x = 0 rsa
    while x+1 ≤ k do x := x+1 od
post x = k
```

Metaprograms are our ultimate targets, and therefore, programmers in **Lingua** will, in fact, develop metaprograms in **Lingua-V**. This language will include five major syntactic categories:

1. *Programs* — that are just programs in **Lingua**.
2. *Conditions* — that express properties of states; their denotations are three-valued partial predicates on states, and they include all boolean expressions of **Lingua**.
3. *Assertions* — that are instructions aborting program executions, if an indicated condition is not satisfied. Syntactically assertions are of the form **asr** con **rsa**, where con is a condition.
4. *Specified programs* — that are programs with nested assertions.
5. *Metaprograms* — that are specprograms with pre- and postconditions.

Lingua-V is, in a sense, a metalanguage since it is used to talk about programs in **Lingua**. The denotations of metaprograms are just classical truth values tt or ff, which means that metaprograms are simply correct or not. An important consequence of this fact is that at the level, where we talk about the development of correct metaprograms, we use classical two-valued logic.

> *In developing correct programs, we remain in the world of classical two-valued logic.*

The fact that in **Lingua-V** we use 3-valued conditions may lead to a false conclusion that the development of correct metaprograms must be carried in a 3-valued logic[73]. Note, however, that our conditions constitute just a category of expressions, which only "happen to look like" logical formulas, but at the level of program development they are not.

To formulate the rules of constructing correct metaprograms, we shall need yet another metalevel. We denote it by **Lingua-MV** and assume that it includes **Lingua-V** plus the following syntactic categories:

1. *Patterns* — that describe sets of elements of **Lingua-V**, e.g. a pattern of a metaprogram may be of the form **pre** prc **:** spr **pos** poc, where prc and poc are metavariables running over conditions and spr is a metavariable running over specified programs.
2. *Metaconditions* — that describe properties of conditions or their patters, i.e. that one condition is stronger than another one.
3. *Metaprograms* — that describe properties of programs or of their patterns.
4. *Metaprogram construction rules* that belong to two categories:
    a. *nuclear rules* — assuring that metaprograms matching certain patterns are correct,
    b. *implicative rules* — assuring that if some metaconditions and/or metaprograms are true/correct, that some other metaprograms are correct.

An example of a nuclear rule may be the following:

```
pre (ide is free) and (tex is type)
   let ide be tex with yex tel                                        (9.1-1)
post var ide is tex with yex
```

It expresses the fact that for any

```
ide  : Identifier,
tex  : TypExp,
yex  : YokExp
```

metaprograms matching pattern (9.1-1) are correct. From this rule we may derive the following concrete correct metaprogram:

```
pre (length is free) and (real is type)
   let length be real with value > 0 tel
post var length is real with value > 0
```

In turn, an example of an implicative rule may be the following:

pre prc : spr **post** poc	*metaprogram pattern*
poc ⇨ poc-1	*metacondition pattern*
pre prc : spr **post** poc-1	*metaprogram pattern*

This rule ensures that for any prc, spr, poc and poc1 (of appropriate categories) if both propositions above the line are true, then the metaprogram below the line is correct. It is to be emphasized that in our approach every construction rule is a theorem — rather than an axiom of a logic of programs — and therefore must be proved. In our approach, we do not develop any "logic of programs" as in the approaches of C.A.R. Hoare [61], [5] and [6] or E. Dijkstra [50] and [51], or as in algorithmic logic [10].

To simplify our wording we shall informally identify patterns with syntactic elements that they represent. E.g. we will say that (9.1-1) is a metaprogram, rather than a pattern of a metaprogram.

To conclude this section let's formulate some remarks about the degree of formalization of our linguistic levels:

---

[73] Readers interested in an analysis of a variety of 3-valued logics that may be based on our 3-valued predicates, are referred to [65].

1. **Lingua** is a programming languages, which has been fully formalized, i.e. it has a formally defined syntax and semantics. So far it hasn't been completely described — some definitions of its elements have been skipped — but its definition must be completed before the implementation of the language.
2. **Lingua-V** is a metalanguage that we shall not formalize at the moment but, again, it should be formalized and completed in the future when it comes to the development of a computer support for the development of correct metaprograms (see Sec. 13.1).
3. The situation with **Lingua-MV** is analogous. We won't formalize it now, but hopefully some of our readers will do it in the future.
4. **MetaSoft** is our meta metalanguage that we use to talk about all our languages. This metalanguage will not be formalized.

Although **Lingua** was developer from denotations to syntax, **Lingua-V** and **Lingua-MV** will be developed in a converse order since in this case:

- we are not formalizing their definitions,
- we are developing them by extending an existing language **Lingua**.

It should be mentioned at the end that there is one more language to be formalized in the future — a language for the development of denotational definitions of programming languages. So far **Lingua** has been defined in a non-formalized **MetaSoft** but in the future one may think of developing a computer system supporting language designers developing denotational definitions of new languages. In such a case a new metalanguage will be necessary. We briefly discuss this issue in Sec. 13.2.

9.2 Conditions

9.2.1 General assumptions about conditions

Denotationally *conditions* represent partial functions from states to boolean values or errors:

cod : ConDen = WfState → BooValE the denotations of conditions

For future use we introduce the following notations for truth values

tv = (tt, 'boolean')
fv = (ff, 'boolean')

By

con : Condition = …

we shall denote the (colloquial) syntactic domain of conditions. As metavariables running over Condition we shall also use

- prc to denote *preconditions*,
- poc to denote *postconditions*.

The syntactic domain of conditions of a "practical language" may be very large, and strongly dependent on the domain of applications of such a language. Therefore, we shall not attempt to define a "complete" language of conditions. Instead we only list basic assumptions about this language, and we show its main categories.

Our first assumption is that the domain of conditions is closed under 3-valued logical connectives and quantifiers, i.e.:

(con1 **and** con2) , (con1 **or** con2) , (**not** con) , (∀ ide : con) | (∃ ide : con)

belong to Condition for any con1, con2, con : Conditions and ide : Identifier. To gain the commutativity of conjunction and disjunction we assume that the boolean constructors are defined in the Kleene's style ra-

ther than in the style of McCarthy's, (Sec. 2.10)[74]. We also assume that we may skip parentheses in a usual way.

The semantics of conditions will be denoted be square brackets [ ], and besides we also introduce the concept of a *truth domain* of a condition:

[con] : WfState → BooValE semantics of conditions
{con} = {sta | [con].sta = tv} truth domains of a conditions

From now on we shall use square brackets to denote the semantics of all components of metaprograms, assuming that a context will always indicate which semantics we mean. We assume further that conditions should be error transparent (cf. Sec. 2.9), i.e., that for any condition con

if is-error.sta then [con].sta = error.sta.

In the end we assume that Condition includes a special condition **NF** that is never false, i.e., such that

[**NF**].sta =
is-error.sta → error
true → tv

Note that we can't introduce a conditions that is always true, because it would be not error transparent.

9.2.2 Value-oriented conditions

Value-oriented conditions describe the properties of values assigned to variables and attributes in states. We assume that syntactically they include all value expressions with boolean values, i.e. boolean expressions, such as, e.g.,

x+1 < 2*z **and** z > 0,

but we assume that their boolean connectives are understood in Kleene's way (cf. Sec. 9.2). Value-oriented conditions may also include conditions that are not boolean expressions. Typical examples in this category are equality conditions of the form

vex-1 **=** vex-2.

Note that at the level of boolean expressions, we usually do not allow comparisons of structural values, such as e.g., lists, arrays, objects or databases, since this might be computationally too expensive. However, we allow such comparisons at the level of conditions, because in this case we do not check (compute) the equalities, but we only use them to express the properties of programs. Another example of a condition that is not a boolean expressions may be

increasingly ordered real (ide)

This condition is satisfied if ide points to a list of real numbers ordered increasingly

9.2.3 Cov-oriented conditions

The mechanism of type-covering relations imposes a necessity of checking types' compatibilities in four following situations:

1. when a value is assigned to a reference by an assignment instruction,
2. when a value of an actual parameter is assigned to the reference of a formal parameter by the mechanism of entering a procedure call,
3. when a reference of an actual parameter is assigned to the corresponding formal parameter by the mechanism of entering a procedure call,
4. when a reference of a formal parameter is passed (back) to the corresponding actual parameter.

---

[74] In building the denotations of boolean value-expressions we still use McCarthy's logical connectives.

In cases 1., 2. and 3 we refer to the current covering relation, but in case 4., i.e., at the exit of a procedure, we have to refer to the covering relation of a call-time state (Sec. 6.6.3.5).

To illustrate this case assume that at the exit of an imperative-procedure call we have a local terminal state (see Sec. 6.6.3)

lt-sta = ((lt-cle, lt-pre, lt-cov), lt-sto)

with the following bindings of a formal reference-parameter ide-fr:

ide-fr → (tok, (typ-r, yok, ota)) → (dat, typ-v).

Since lt-sta is well-formed, the following relationship is satisfied.

typ-r **TTA.lt-cov** typ-v. (9.2.3-1)

Now, the mechanism of returning the reference of ide-fr to an actual reference-parameter ide-ar is activated, and creates a global terminal state

gt-sta = ((gt-cle, gt-pre, gt-cov), gt-sto).

with the following bindings:

ide-ar → (tok, (typ-r, yok, ota)) → (dat, typ-v).

Since the operation of returning parameters must guarantee the well-formedness of gt-sta, we have to check if the following relationship is satisfied:

typ-r **TTA.gt-cov** typ-v.

However, as we have seen in Sec. 6.6.3.5, the global terminal covering relation cov-gt is equal to a call time covering relation cov-ct, which means that we must ensure the relationship

typ-r **TTA.ct-cov** typ-v, (9.2.3-2)

at the exit of the body of our procedure. Note in this place that lt-cov may be larger than ct-cov — since it might have been enriched during the execution of the procedure's body — and therefore (9.2.3-1) may be satisfied, whereas (9.2.3-2) is not.

To check the satisfaction of (9.2.3-2) at the exit of the procedure's body in the rule of the development of a procedure (Rule 9.4.6.3-1), we have to express this fact in the postcondition of the body, hence as a property of the local terminal state. But in this state we do not "have access" to the call-time relation. Consequently, we have to somehow "memorize" ct-cov in the syntax of conditions.

To cope with this problem we first introduce a concept of *cov-expressions* that evaluate to covering relations:

```
coe: CovExp =
        (TypExp , TypExp)                |
        (TypExp , TypExp) ; CovExp
```

Their semantics is the following:

[coe] : WfState ⟼ CovRel | Error

We skip its obvious definition, assuming that an error message is signalized in three situations:

1. if some involved type expressions evaluate to errors,
2. if in a pair of types one is a data type and the other is an object type,
3. if an object type is not a name of a declared class.

Now, to express the satisfaction of (9.2.3-2) as a property of a local terminal state, we introduce two new categories of conditions:

coe **is current** — coe evaluates to the current covering relation
fpa **well-valued in** coe — references of formal parameters fpa accept their values wrt coe

The denotation of the first condition is the following:

```
[coe is current].sta =
    is-error.sta      ➔ error.sta
    let
        c-cov                   = [coe].sta
        ((cle, pre, cov), sto)  = sta
    c-cov : Error  ➔ c-cov
    c-cov ≠ cov    ➔ 'current cov-relation not confirmed'
    c-cov = cov    ➔ tv
```

To define the denotation of the second one we need an auxiliary function

list-of-ide : ForPar ⟼ LisOfIde

that given a (list of) formal parameter, i.e., a syntactic element, e.g.,:

x, y, z **as real,** n, m, p **as integer**

returns the list of identifiers

x, y, z**,** n, m, p

(cf., a similar construction in Sec. 6.6.3.3). We skip a formal definition of this function.

```
[fpa well-valued in coe].sta =
    is-error.sta                                  ➔ error.sta
    let
        cov              = [coe].sta
        (ide-1,…,ide-n)  = list-of-ide.fpa
    cov : Error                                   ➔ cov
    let
        (env, (obn, dep, ota, sft, 'OK')) = sta
    obn.ide-i = ?                                 ➔ 'variable not declared'      for i = 1;n
    dep.(obn.ide-i) = ?                           ➔ 'variable not initialized'   for i = 1;n
    (∀i)(obn.ide-i VRA.cov dep.(obn.ide-i))       ➔ tv
    true                                          ➔ fv
```

Note that the acceptance of values by corresponding references is checked wrt the type-covering relation indicated by coe which needs not coincide with the current relation carried by sta.

Given these new categories of conditions we can express the fact[75] that at the exit of procedure body formal reference parameters are well-valued wrt a call-time covering relation:

```
prc-call   ⇨ coe is current                 and
poc-body   ⇨ fpa-r well-valued in coe
```

This technique will be used in Sec. 9.4.6.3 where we formulate a rule for the creation of procedure declarations that lead to correct procedure calls.

Our last condition in this section concerns the enrichment of a covering relation by a new pair of types. We recall (cf. Sec. Sec. 5.4.2 and 6.7.5) that two types may be added to a covering relation if:

1. they are different,
2. they do not belong to this relation,
3. they are either both data types or both object types,
4. if they are object types, i.e., identifiers, then they have to be the names of declared classes.

The following condition checks if a given pair of types can be added to a current covering relation:

[75] This fact is a metacondition from the level of **Lingua-MV** (see Sec. 9.1).

```
[ consistent(tex-1, tex-2) ].sta =
   is-error.sta                          ➔ error.sta
   [tex-i].sta : Error                   ➔ [tex-i].sta     for i = 1,2
   let
      typ-i                  = [tex-i].sta          for i = 1,2
      ((cle, pre, cov), sto) = sta
   typ-1 : DatTyp and typ-2 : ObjTyp     ➔ 'types not comparable'
   typ-2 : DatTyp and typ-1 : ObjTyp     ➔ 'types not comparable'
   (typ-1, typ-2) : cov                  ➔ 'types already in covering relation'
   typ-1, typ-2 : DatTyp                 ➔ tv
   cle.typ-i = ?                         ➔ 'object types must be declared' for i = 1,2
   true                                  ➔ tv
```

9.2.4 Value-, type- and reference-oriented conditions

Conditions of this category describe, except (11), properties of output states of non-procedural categories of declarations. Condition (11) does not belong to this group, but is listed here because it is a necessary prerequisite for all declarations to execute cleanly. Below we show some typical examples of conditions associated with values, types and references, excluding these of Sec. 9.2.3:

(1) ty-ide **is type in** cl-ide, ty-ide is declared as type constant in class cl-ide
(2) ide **is** tex, ide is a type constant pointing to a type indicated by tex
(3) **att** at-ide **is** tex **with** yex **in** cl-ide **as** pst, at-ide is declared with tex and yex in class cl-ide…
(4) **var** ide **is** tex **with** yex, ide is a declared variable of type tex and yoke yok
(5) rex **is reference**, reference expression rex evaluates cleanly
(6) vex **is value**, value expression vex evaluates cleanly
(7) tex **is type,** type expression tex evaluates cleanly
(8) cli **is class,** cli is either empty-class or an identifier of a declared class
(9) ide **child of** cli, ide is an identifier of a declared class which is a child of class indicated by cli
(10) tex1 **covers** tex2, tex1 and tex2 evaluate cleanly and…
(11) ide **is free** ide has not been declared

Below we define only two of these categories of conditions since the remaining ones seem obvious.

```
[ att at-ide is tex with yex in cl-ide as pst ].sta =
   is-error.sta                     ➔ error.sta
   [tex].sta : Error                ➔ [tex].sta
   let
      ((cle, pre, cov), sto) = sta
      typ                    = [tex].sta
      yok                    = [yok]         we recall that the denotation of a yoke expression is a yoke
   cle.cl-ide = ?                   ➔ 'class unknown'
   let
```

```
        (cl-ide, tye, mee, obn) = cle.cl-ide
    obn.at-ide = ?                    ➔ 'attribute unknown'
    let
        (tok, (at-typ, at-yok), at-ori) = obn.at-ide
    typ ≠ at-typ                      ➔ 'types not compatible'
    yok ≠ at-yok                      ➔ 'yokes not compatible'
    pst = 'private' and ar-ori ≠ cl-ide ➔ 'privacy status not adequate'
    pst = 'public' and ar-ori ≠ $     ➔ 'privacy status not adequate'
    true                              ➔ tv
```

Note that in yok ≠ at-yok we compare two functions, which is not computable, but this fact does not matter, since conditions are not evaluated.

Proceeding to the definition of (9) we recall that a child class named ch-ide is a child of a parent class named pa-ide, if it inherits all signatures and all attributes of the parent class.

```
[ch-ide child of pa-ide].sta =
    is-error.sta               ➔ error.sta
    let
        ((cle, pre, cov), sto) = sta
    cle.pa-ide = ?             ➔ 'parent class unknown'
    cle.ch-ide = ?             ➔ 'child class unknown'
    let
        pa-cla   = cle.pa-ide
        (ch-ide, [ ], fu-mee, ch-obn)        = make-funding-class.ch-ide. pa-cla      (see Sec. 6.7.3)
        (ch-ide, ch-tye, ch-mee, ch-obn)     = cle.ch-ide
    fu-mee /⊆ ch-mee               ➔ 'signatures not compatible'
    dom.pa-obn /⊆ dom.ch-obn       ➔ 'attributes not compatible'
    true                           ➔ tt
```

At the end a comment about condition (6). For it to be satisfied, all variables and attributes in vex must be declared and initialized. This is why (6) has been classified as declaration-oriented condition. It is also worth noticing that the satisfaction of (6) implies that the evaluation of vex won't generate an error message, e.g., an overflow.

9.2.5 Procedure-oriented conditions

Conditions of this category describe the effects of procedure declarations (operator @ is defined in Sec. 9.2.7).

(1) pr-ide (val fpv ref fpr) begin body end imperative in cl-ide,
(2) fu-ide (val fpv ref tex) begin body return vex end functional in cl-ide,
(3) ob-ide (val fpv ref ob-ide) begin body end objectional in cl-ide,
(4) procedure cl-ide.pr-ide opened
(5) (pass actual val apa-v ref apa-r to formal val fpa-v ref fpa-r with cl-ide) @ con

The denotation of (1) is the following:

```
[pr-ide (val fpc-v ref fpc-r) begin body end imperative in cl-ide].sta =
    is-error.sta                          ➔ error.sta
    let
        ((cle, pre, cov), sto) = sta
    cle.cl-ide = ?                        ➔ 'class unknown'
    let
        (cl-ide, tye, mee, obn) = cle.cl-ide
    mee.pr-ide = ?                        ➔ 'pre-procedure unknown'
    let
```

declared-pre-proc = mee.pr-ide
expected-pre-proc = create-imp-pre-pro.([fpd-v], [fpd-r], [body])
declared-pre-proc ≠ expected-pre-proc ➔ fv
true ➔ tv

Our condition claims three facts:

1. cl-ide is a name of a declared class,
2. pr-ide is a name of a procedure in this class,
3. pre-procedure pointed by pr-ide is equal to a pre-procedure that would be created by a declaration **proc** pr-ide **(val** fpc-v, **ref** fpc-r**)** **begin** body **end**.

Note that in 3. we do not claim that the body of the declared pre-procedure is body, but that the declared procedure (a denotational element) is identical with a procedure generated by **proc** pr-ide **(val** fpc-v, **ref** fpc-r**)** **begin** body **end**. It is, therefore, not a claim about syntax, but about its "denotational effect". This condition is also not computable. The definitions for cases (2) and (3) are analogous.

Condition (4) claims that pre-procedure pr-ide declared in class cl-ide gave rise — due to the opening declaration — to a procedure assigned to procedure indicator (cl-ide, pr-ide) in the procedure environment of the current state. We skip an obvious definition.

The last category of conditions has an algorithmic character (see Sec. 9.2.7), and will be used to describe the effect of an action of passing actual parameters to formal parameters in a procedure call. In its definition we shall refer to function pass-actual defined in Sec. 6.6.3.4. To define this condition we only need to define its imperative component:

[**pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** cl-ide] : WfState ⟼ WfState

[**pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** cl-ide].sta =
is-error.sta ➔ sta ◄ error.sta
let
(env, sto) = sta
new-sto = pass-actual.(fpa-v, fpa-r, apa-v, apa-r, cl-ide).env.sto
is-error.new-sto ➔ sta ◄ error.new-sto
true ➔ (env, new-sto)

We recall that cl-ide is a name of a class.

A state that satisfies condition (5) guarantees that starting with it, the execution of pass-actual will terminate cleanly, and the output state will satisfy condition con. We shall use this condition in Sec. 9.4.6.3, where we formulate a rule for constructing correct procedure calls.

9.2.6 Assertions and specified programs

As we have already seen in Sec. 8, the rules of the development of correct metaprograms are lemmas which guarantee the correctness of some metaprograms provided that their components were correct. In an abstract case where programs are represented by binary relations between (abstract) states, correctness of programs was expressed by pre- and postconditions. However, at the level of a programming language sometimes we may need to talk not only about the properties of input and output states of programs, but also about properties of their intermediate states.

We shall start from the introduction of a syntactic category of *assertions* whose domain is defined by the following equation:

asr : Assertion = **asr** Condition **rsa**

The semantic of assertions is the following:

[asr] : WfState → WfState
[**asr** con **rsa**].sta =
is-error.sta ➔ sta

[con].sta = ? ➔ ?
[con].sta : Error ➔ sta ◄ [con].sta
[con].sta = fv ➔ sta ◄ 'assertion not satisfied'
true ➔ sta

Note that an error message will be generated by assertions in two situations:

1. when the value of the condition is an error,
2. when the condition is not satisfied.

An assertion may be regarded as a filter that is transparent for states satisfying the included condition, and otherwise aborts the execution of a program.

Another new concept that we shall need in the future will be used in building rules for the development of class declarations (Sec. 9.4.4.2). By an *anchored class transformer* we mean an imperative element with the following syntactic domain:

act : AncClaTra = ClaTra **in** Identifier

and the following semantics:

[ctr **in** ide] : WfState → WfState
[ctr **in** ide] = [ctr].ide.

The identifier in this structure is called an *anchor*. We recall that the denotations of class transformers constitute the following domain:

ctc : ClaTraDen = Identifier ⟼ WfState → WfState.

which means that a transformer, when given a (class) identifier, becomes a state-to-state function, i.e., a denotation of an anchored class transformer.

Now we are ready to define *specified programs* and their specified components. Intuitively they are imperative components of programs with nested assertions. Below we define the corresponding syntactic domains. We also introduce the concepts of *on-zones* and *of-zones* of specinstructions with semantics defined a little later.

sin : SpeIns = *specified instructions* or *specinstructions*
Instruction |
Assertion |
SpeIns **;** SpeIns |
asr con **in** SpeIns **rsa** | *on-zones*
off con **in** SpeIns **ffo** | *off-zones*
if ValExp **then** SpeIns **else** SpeIns **fi** |
if-error ValExp **then** SpeIns **fi** |
while ValExp **do** SpeIns **od** |
skip-ins

sde : SpeDec = *specified declarations* or *specdeclarations*
Declaration |
Assertion |
SpeDec **;** SpeDec |
skip-dec

sct : SpeClaTra = *specified class transformers* or *spectransformers*
AncClaTra |
Assertion |
SpeClaTra **;** SpeClaTra |
skip-sct

spp : SpeProPre = *specified program preambles*
SpeDec |

SpeIns |
SpeProPre ; SpeProPre |
skip-spp

spr : SpePro = *specified programs* or *specprograms*
SpeProPre **; open procedures ;** SpeIns |
SpeProPre |
SpeClaTra

Sometimes we shall need to express the fact that an assertion **asr** con **rsa** is to be satisfied not only in one particular location of a specinstruction sin, but during the "whole execution" of sin, i.e., by all its intermediate states with the exclusion of local states of procedure calls. In such a case, instead of "physically" inserting an assertion into sin in all expected positions, we shall use a *on-zone instruction* of the form

asr con **in** sin **rsa**

where con will be called a *zone assertion.* Typical situations where we want to insure the satisfaction of an assertion in a zone take place in data-base programming, where zone assertions are known as *integrity constraints*. In the same area of applications we sometime wish to "switch off" a zone assertion, to perform an operation that temporarily "spoils" an integrity constraint. In such a case we shall write

off con **in** sin **on**

to indicate the range of the off-zone. Examples of the use of both concepts in metaprogram derivation are shown in Sec. Sec. 9.5.1 and 9.5.1.

We shall not formalize the ranges of assertions, since this would lead to too many technicalities, e.g., in the case when zone ranges overlap. We only wish to signalize the idea, and to use it in simple situations.

In the end it is worth noticing that the semantics of zones is not compositional, since in a general case the denotation

[**asr** con **:** sin **rsa**]

can't be described as a function of [sin] and [con]. As an example consider two following specinstructions:

asr x > 0: x := x **rsa** and
asr x > 0: x := -x ; x := -x **rsa**

The denotations of their instructions are identical, but the denotations of declarations are not.

9.2.7 Algorithmic conditions

Algorithmic conditions are conditions that include specified programs. The domain of *algorithmic conditions* is defined in the following way:

con : AlgCondition =
SpePro **@** Condition | *left-algorithmic conditions*[76]
Condition **@** SpePro *right-algorithmic conditions*

The semantics of algorithmic conditions is as follows:

[spr **@** con].sta =
(∃ sta1 : {con}) [spr].sta = sta1 ➔ tv
true ➔ fv

[con **@** spr].sta =

[76] Left algorithmic conditions, although not called in this way, constituted a fundament of *algorithmic logic* developed at Warsaw University in the decades 1970. and 1980. (see [10]**Błąd! Nie można odnaleźć źródła odwołania.**).

(∃ sta1 : {con}) [spr].sta1 = sta ➔ tv
true ➔ fv

Note that in the first case, since sta1 : {con} and all conditions are error transparent by definition, sta1 can't carry an error. This means that spr with input sta terminates cleanly. Also sta can't carry an error since specified programs are error transparent.

The situation in the second case is different. Here we start from an input state that satisfies con, and therefore must be error free, but terminal states may carry errors.

Since algorithmic conditions are two-valued[77] they are unambiguously identified by their truth domains. Consequently their equivalent definitions are following :

{spr @ con} is the set of all input states that cause spr to terminate cleanly with output states that satisfy con,
{con @ spr} is the set of all output states of spr for input states that satisfy con.

Algorithmic condition may be not computable since the termination property of programs is not decidable. The following obvious equalities also hold (for the definition of '●' see Sec. 2.7):

{spr @ con} = [spr] ● {con}
{con @ spr} = {con} ● [spr]

At the end we assume that the domain Condition is closed under both operations of building algorithmic conditions. It means, in particular, that conditions in algorithmic conditions may be algorithmic themselves.

9.3 Metaconditions

9.3.1 Basic categories of metaconditions

Metaconditions describe semantic properties of conditions, specified programs and their components. Syntactically, they do not belong to the validating language **Lingua-V** but to the metalanguage **Lingua-MV** (cf. Sec. 9.1). Each metacondition is either true or false, which means that the denotations of metaconditions are classical logical values tt or ff.

We assume that the language of metaconditions will be closed under classical connectives and, or, not and implies. Atomic metaconditions, and their meanings, are defined as follows:

con-1 ⇨ con-2	iff(def)	{con-1} ⊆ {con-2}	stronger than; metaimplication
con-1 ⬄ con-2	iff(def)	{con-1} = {con-2}	weakly equivalent
con-1 ⊑ con-2	iff(def)	[con-1] ⊆ [con-2]	less defined
con-1 ≡ con-2	iff(def)	[con-1] = [con-2]	strongly equivalent

The relations, i.e., ⇨, ⊑, ⬄ and ≡ will be called *metapredicates.* To better understand their nature let's see the following examples, where we assume that the evaluation of the square root $\sqrt[2]{x}$ generates an error if x is not a nonnegative real number:

x>0 **and** $\sqrt[2]{x} > 2$	≡	x > 4	if x is not a real-number variable, then both sides generate the same error,
$\sqrt[2]{x} > 2$	⬄	x > 4	but ≡ does not hold,
$\sqrt[2]{x} > 4$	⇨	x > 3	but neither ⬄ nor ⊑ holds.

If we assume that for negative x function $\sqrt[2]{x}$ is undefined rather than generates and error, then the following relation holds:

$\sqrt[2]{x} < 2$	⊑	x < 4	but neither ≡ nor ⬄ holds,

[77] We could have made them three-valued, but we do not need to do so, since in using algorithmic conditions we shall refer to their truth domains only.

The following rather obvious relationships hold between metapredicates[78]:

con1	≡	con2	is equivalent to	con1 ⊑ con2 **and** con2 ⊑ con1
con1	⇔	con2	is equivalent to	con1 ⇒ con2 **and** con2 ⇒ con1
con1	≡	con2	implies	con1 ⇔ con2
con1	≡	con2	implies	con1 ⊑ con2
con1	⇔	con2	implies	con1 ⇒ con2

It is important to understand the difference between three implication-like constructors that belong to three different logical and linguistical levels:

1. **implies** : Condition x Condition ⟼ Condition — implication in **Lingua-V**,
2. ⇒ : Condition x Condition ⟼ {tt, ff} — metaimplication in **Lingua-MV**,
3. implies : {tt, ff} x {tt, ff} ⟼ {tt, ff} — (usual) implication in **MetaSoft**

Using metaimplications and algorithmic conditions, we can easily express the total and the partial correctness of a specprogram spr for a precondition prc and a postcondition poc:

prc ⇒ spr @ poc clean total correctness
prc @ spr ⇒ poc partial correctness

As we see, spr @ poc is the *weakest total precondition* for spr and poc, and prc @ spr is the *strongest partial postcondition*[79] for spr and prc .

In our future rules of the development of correct programs we shall use another category of **Lingua-MV** called *metaprograms* that express of specified programs and are of the form

pre prc **:** spr **post** poc

Their meaning is defined in an obvious way:

pre prc **:** spr **post** poc **iff (def)** prc ⇒ spr @ poc

Note that since our conditions have been assumed error-transparent (Sec. 9.2.1), clean total correctness insures non-abortion.

In an analogous way we define the categories of *metainstructions*, and *metadeclarations*, and, in general, *metacomponents of specprograms.*

The notion of total correctness with clean termination was defined by Andrzej Blikle in [28]. It is different from total correctness considered by other authors (cf. [4],[5], [6], [8], [10], [50] or [51]), where programs never generate errors, i.e., never abort. In the evaluation of our programs, error messages may be raised, but if a program is correct, this will not happen.

A special category of metaconditions will be used in the development of **while-do-od** instructions (Sec. 9.4.6) and include metacondition of the form:

limited replicability of sin **if** con

This metacondition is true if there is no infinite sequence of states sta-1, sta-2,… such that for all i = 1,2,…

[con].sta-i = tv
sta-(i+1) = [sin].sta-i

Cf. limited replicability of a function defined in Sec. 8.7.2.

---

[78] It is worth noticing that on the ground of our non-classical calculus of conditions we have two concepts of satisfiability — *strong satisfiability* (con ≡ TT) con is always true, and *weak satisfiability* (con ⊑ TT) con is never false. Readers interested in logics based on these concepts are referred to [65].

[79] These concepts are due to Edsger W. Dijkstra (see [50] and [51]).

9.3.2 Properties of metapredicates

This section includes a list of lemmas which are useful in the development of correct metaprograms.

Lemma 9.3.2-1 *Relations ≡ and ⇔ are both equivalences, i.e., they are reflexive, symmetric, and transitive.*

Lemma 9.3.2-2 *Strong equivalence is a congruence wrt* **and**, **or** *and* **not**, *i.e., the replacement of a subcondition of a condition by a strongly equivalent one result a condition strongly equivalent to the initial one.*

Lemma 9.3.2-3 *Weak equivalence is a congruence wrt* **and** *and* **or**.

Weak equivalence is not a congruence wrt negation which means that

con1 ⇔ con2 does not imply **not** con1 ⇔ **not** con2

For instance, although

$\sqrt[2]{x} > 2$ ⇔ x > 4

is satisfied, the metacondition

$\sqrt[2]{x} \leq 2$ ⇔ x ≤ 4

is not, since for x = −1 the right-hand-side equation evaluates to tv, but on the left-hand side, we have an error.

Lemma 9.3.2-4 *The operators* **and** *and* **or** *are strongly associative, i.e.*

(con1 **and** con2) **and** con3 ≡ con1 **and** (con2 **and** con3)
(con1 **or** con2) **or** con3 ≡ con1 **or** (con2 **or** con3)

Of course, they are also weakly associative since strong equivalence implies weak equivalence.

Lemma 9.3.2-5 *The operator* **and** *is strongly left-hand-side distributive wrt to* **or** *and vice versa, i.e..*

con1 **and** (con2 **or** con3) ≡ con1 **and** con2) **or** (con1 **and** con3)
con1 **or** (con2 **and** con3) ≡ con1 **or** con2) **and** (con1 **or** con3)

However, both operators are not strongly right-hand-side distributive. Indeed (not quite formally written):

(tv **or** ee) **and** fv = fv but (tv **and** fv) **or** (ee **and** fv) = ee
(fv **and** ee) **or** tv = tv but (fv **or** tv) **and** (ee **or** tv) = ee (9.3.2-1)

Lemma 9.3.2-6 *The operator* **and** *is weakly left-hand-side distributive wrt* **or** *i.e.*

(con1 **or** con2) **and** con3 ⇔ (con1 **and** con3) **or** (con2 **and** con3)

However, **or** is not even weakly left-hand-side distributive wrt **and** which can be seen in (9.3.2-1).

Lemma 9.3.2-7 *The de Morgan's laws for* **and** *and* **or** *and for the negation of quantifiers are satisfied with strong equivalence.*

Lemma 9.3.2-8 *Conjunction is weakly commutative, i.e.,*

con1 **and** con2 ⇔ con2 **and** con1

However, conjunctions are not strongly commutative, and the disjunction is not even weakly commutative, since:

tv **or** ee = tv but ee **or** tv = ee

Lemma 9.3.2-9

If con1 ⇒ con2 then con1 **and** con2 ≡ con1.

Besides the two-argument metapredicates, we also define three-argument metapredicates which will be used in the development of correct metaprograms:

con1 ≡ con2 **whenever** con means con **and** con1 ≡ con **and** con2
con1 ⇔ con2 **whenever** con means con **and** con1 ⇔ con **and** con2

con1 ⇨ con2 **whenever** con   means con **and** con1 ⇨ con **and** con2

In all these cases, we say that con constitutes a *logical context* or simply a *context* for the metapredicate that follows. We shall also say that the *equivalence* con1 ≡ con2 *is satisfied under the condition* con and analogously for a weak equivalence and metaimplication. E.g. the following metapredicates are satisfied:

n > x^2 ≡ $\sqrt[2]{n}$ > x **whenever** (n ≥ 0 **and** x ≥ 0)
n > x^2 ⇔ $\sqrt[2]{n}$ > x **whenever** x ≥ 0

The context is usually a condition in whose range we want to replace one condition by another one.

All considerations presented here were published by A. Blikle in the decade 1980 in [28] and [31], and the development of these ideas towards three-valued deductive theories was investigated in a paper [65] by B. Konikowska, A. Tarlecki and A.Blikle.

9.3.3 Metaconditions associated with programs

As we are going to see in Sec. 9.4.1, in the development of correct metaprograms the development of pre- and postconditions is equally vital as the development of specprograms. To systematise the development of conditions we shall define three groups of metaconditions depending on specprograms, metaprograms and specification languages respectively. The first group describes *behavioural properties* of conditions versus specprograms:

con **resilient to** spr	if	con **@** spr ⇨ con	—	con *is resilient* to spr, if its satisfaction is not violated by spr,
con **consumed by** spr	if	con ⇨ spr **@ not** con	—	con is *consumed by* spr, if it is like a raw material that assures execution but disappears after it,
con **catalyzing for** spr	if	con ⇨ spr **@** con	—	con is *catalysing for* spr, if it is like a chemical catalyzer — it assures execution but is not consumed,
con **essential for** spr	if	con ≡ spr **@ NF**	—	con is *essential for* spr if it is the weakest preconditions that ensures a clean termination of spr.

Since specprograms are by definition deterministic (represent functions), catalyzing conditions are resilient, but not vice versa. To illustrate the defined metaconditions consider a simple metaprogram:

```
pre (x is free) and (var y is integer with value < 3) :
    let x be real with value > 10 tel;
    x := 17,3
    ...
post (var x is real with value > 10) and (var y is integer with value < 3) and (x = 17)
```

The following relations hold:

- **var** y **is** integer **with value** < 3 is **resilient** to the declaration of x, but not catalyzing,
- x **is free** is **consumed** by the declaration of x and is essential for it.
- **var** x **is** real **with value** > 10 is **catalyzing** for x := 17,3.

Note that the catalyzing condition for the assignment is not essential, since is not the weakest. The essential condition for x := 17,3 is **var** x **is** real, i.e., with a trivial yoke.

It is to be emphasized that although we considered a metaprogram in our example, all illustrated properties concern relations between a condition and a specprogram, and do not depend on the fact that our exemplary metaprogram is correct.

Our second group of metaconditions concerns the satisfaction of conditions against the executions of correct metaprograms, i.e., against the sequences of consecutive states of such executions. To avoid talking about sequences of states, that would lead to an alternative semantics of programs[80], we introduce an auxiliary concept of a *cut* of a specprogram.

Let AlpLin-V be an alphabet of **Lingua-V**, i.e., a finite set of characters such that all metaprograms are words over AlpLin-V. Let

phr : Phrase = AlpLin-V*

be the set of all words over this alphabet that we shall call *phrases*. By a *cut* of a metaprogram

mpr = **pre** prc **:** spr **post** poc

we mean any pair of phrases of the form (**pre** prc **:** pre, pos **post** poc), called respectively the *head* and the *tail* of this cut, such that:

pre prc **:** pre **;** pos **post** poc = mpr.

and the semicolon is not "located" in a body of a procedure declarations (we skip a formal definition of "location"). Intuitively, cuts identify "global" semicolons in metaprograms. Note that we do not exclude cuts through class declarations or structured instructions, but we exclude cuts though procedure bodies.

It is evident that cuts of a given metaprogram may be linearly ordered by a relation *earlier/later*. We skip its formal definition. We say that a condition con is *satisfied in cut* (**pre** prc **:** pre, pos **post** poc) if the metaprogram:

pre prc **:** pre **; asr** con **rsa ;** pos **post** poc

is correct. Note that in such a case mpr must be correct as well.

Having defined cuts, we are ready to define *temporal properties* of conditions versus correct metaprograms. Analogously as in the case of behavioural properties we define corresponding metaconditions between conditions and metaprograms. Let

mpr = **pre** prc**:** spr **post** poc

be a correct metaprogram. We say that:

con **primary in** mpr	if	prc ⇨ con, i.e. if con is satisfied at the entrance of the program (and possibly later as well),
con **induced in** mpr	if	there exists a cut of mpr such that con is satisfied in this cut; an induced condition must be eventually satisfied,
con **hereditary in** mpr	if	con once satisfied in a cut, will be satisfied in all later cuts; note that a condition that is never satisfied is hereditary,
con **va-hereditary in** mpr	if	con once falsified, will be falsified in all later cuts; con is va-hereditary iff **not** con is hereditary
con **perpetual in** mpr	if	con is primary and hereditary at the same time, i.e. if it is satisfied in all cuts of mpr.

[80] Such a semantics was introduced and investigated by Andrzej Blikle in [23].

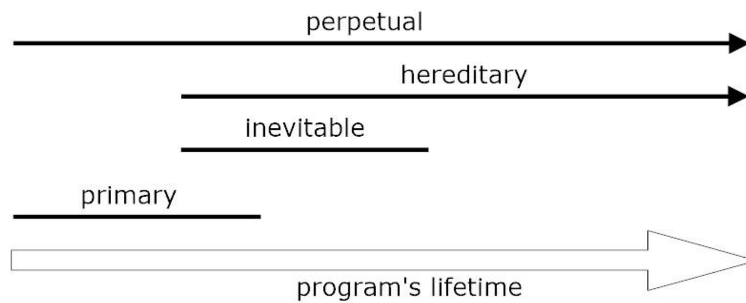


Fig. 9.3-1 Temporal categories of conditions

Note that in all five cases we define a relation between a condition and a correct metaprogram. We do not consider temporal properties of conditions against incorrect metaprograms. To illustrate the introduced concepts let's return to our example of a metaprogram. In this program:

x **is free**	— is primary and va-hereditary,
var y **is** integer **with value** < 3	— is perpetual, since a type once declared remains declared forever,
var x **is** real **with value** > 10	— is induced and hereditary,
x = 17,3	— is induced but not necessarily hereditary.

Note that condition x = 17 maybe in this program hereditary or not, depending whether the value of x is later changed. The situation with **var** x **is** real **with value** > 10 is different. It is hereditary in every correct metaprogram since a variable, once declared, can't be redeclared anymore.

The properties of conditions defined so far describe relations between conditions and spec- or metaprograms, i.e., are program dependent. Our last group of metaconditions describe properties of conditions that are program independent, i.e. satisfied in all programs of a **Lingua-V**.

con **is immunizing**	if	con **hereditary in** mpr for every mpr,
con **is immanent**	if	the value of con is never false, although may be undefined or be an error,
con **is underivable**	if	whenever **pre** prc : spr **pos** poc is correct and poc ⇨ con, then prc ⇨ con.

In **Lingua-V**, typical immunizing conditions are induced by declarations, but if we allow variable redeclarations, they would not be.

Typical immanent conditions describe properties of mathematical beings appearing in our language, such as, e.g.,

x + y = y + x

where + denotes an integer addition.

A condition is underivable, if it can't be induced in a metaprogram, unless it is included in the precondition of this program. Typical underivable conditions in **Lingua-V** are conditions of the pattern ide **is free**[81]. At the same time, since they are essential for declarations, they have to be assumed in the precondition of the program. In practice, in the process of program development, whenever we intend to add a declaration, we have to add an appropriate freeness condition to the precondition of this program, and then to "propagate" it (due to its resilience) to the pre- and postconditions of all preceding programs. More on this issue in Sec. 13.1.

---

[81] We may also think abut "less trivial" underivable conditions such as, e.g., conditions describing properties of databases, that can't be created by **Lingua** programs, but at the same time can be processed by such programs.

In the process of a metaprogram derivation described in Sec. 9.4.1, underivable conditions and hereditary conditions play contrasting roles:

- Whenever we need an underivable condition in a precondition of an intermediate metaprogram, we have to add it to the precondition of the previous program and, therefore, to the precondition of the initial program.
- Whenever we induce a hereditary condition in a postcondition of an intermediate program, we can add it to the postcondition of the next program, and consequently to the postcondition of the final program. We even have to do it, if we want our postcondition to be the strongest one.

As we see, in the process of a metaprogram development we, on one hand, incrementally create the future precondition of this program by adding to it underivable conditions that we shall need later, and on the other — we incrementally create the future strongest postcondition of the program by adding to it hereditary conditions.

At the same time we have "to keep a repository" of immanent conditions that we shall need to prove some facts, e.g., metaimplications, in developing our metaprograms. More about a "logistics" of conditions in Sec. 13.1

9.4 Metaprogram constructions rules

9.4.1 A birds-eye view on a metaprogram development

Due to our assumption in Sec. 6.3 every "completed" metaprogram is of the form

pre prc: spp **; open procedures ;** sin **post** poc

where spp is a specified program preamble and sin is a specified instruction (see Sec. 9.2.6). This form may be "unfolded" to

pre prc **:**
  atp-1 **;** … **;** atp-n ; **open procedures ;** asi-1 **;** … **;** asi-k
post poc

where

- atp-i's are *atomic preambles,* i.e., single declarations of variables or of classes, or atomic specinstructions,
- asi-i's are *atomic specinstructions,* i.e., instructions listed in Sec. 7.3.7, except the last one, plus assertions; note that structured instructions such as **while-do-od** and **if-then-else-fi** are regarded as atomic, although their "internal instructions" may be quite complex.

Consequently, the process of a metaprogram development may be split into a sequence of steps, each building one *atomic metaprogram*:

pre prc-1**:** atp-1 **post** poc-1
pre prc-2**:** atp-2 **post** poc-2
…
pre prc-n **: open procedures post** poc-n
pre prc-(n+1) **:** asi-1 **post** poc-(n+1)
…
pre prc-(n+k+1) **:** asi-k **post** poc-(n+k+1)

where

prc ⇨ prc-1
poc-i ⇨ prc-(i+1) for i = 1,2,…,k+n
poc-(n+k+1) ⇨ poc

In this process we are building not only successive imperative components or our future program, but also successive pre- and postconditions. Observe that although the composition of imperative components may be left to the end of the process, pre- and postconditions have to be build incrementally "as we go". This means that whenever we are creating a next metaprogram

pre prc-i: atp-2 **post** poc-i

we have to insure that the former postcondition metaimplies prc-i. In practice it must be either equal prc-i or of the form val **and** prc-i. Whatever we want to say about programs must be expressed by derivable conditions or included in the precondition of the program. It is why the derivability, resilience and heredity of conditions are such vital issues in program development.

Since atomic metaprograms will be eventually combined into a final metaprogram in using Rule 9.4.2-2, we may restrict our further considerations to the development of atomic metaprograms. Note in this place that atomic metaprograms do not need to be simple. A metaprogram with a single assignment instruction is simple, but developing a while loop or a class declaration with recursive procedures may be quite challenging task (cf. Sec. 9.4.4.2).

We in the sequel shall define three categories of rules for the derivation of correct metaprograms:

- *correctness preserving rules* describing transformations of programs which preserve their correctness but possibly change their meanings, i.e., the denotations of their specprograms (Sec. Sec. 9.4.2 and 9.5),
- *universal rules* concerning all metaprograms (Sec. 9.4.3),
- *specific rules* concerning declarations (Sec. Sec. 9.4.4 and 9.4.5) and instructions (Sec. 9.4.6).

Specific rules may be *atomic*, i.e., claiming an (unconditional) correctness of a metaprogram, or *implicative*, i.e., claiming that if some metaconditions are satisfied, then some metaprograms are correct.

9.4.2 Correctness-preserving modifications of metaprograms

Let's recall (cf. Sec. 9.2.6) that all our specprograms are of the form

spp **; open procedures ;** sin

where spp is a specprogram preamble, and sin is a specinstruction (both may be trivial).

Lemma 9.4.2-1 *If*

pre prc **:** spp **; open procedures ;** sin **post** poc

is correct, then in any execution of spp**;open procedures;** sin *that starts with a state satisfying* prc:

1. *none of* spp, sin, poc *generates an error,*
2. *states in* {prc} *do not bind identifiers that are (going to be) declared in* spp*,*
3. *all assertions in* sin *are satisfied,*
4. *the terminal state does not carry an error.*

Note that **open procedures** never generates an error, and, therefore, we do not need to mention this fact in our lemma.

Lemma 9.4.2-2 *If*

pre prc **:** spp **; open procedures ;** sin **post** poc

is correct and sin1 *has been created from* sin *by the removal of an arbitrary number of assertions or on-assertion-declarations, then the metaprogram*

pre prc **:** spp **; open procedures ;** sin1 **post** poc

is correct as well.

Lemma 9.4.2-3 *The replacement in a correct metaprograms its pre- or post-condition or a condition in an assertion by a weakly equivalent condition, does not violate the correctness of the program.*

For pre- and post-conditions the proof is obvious. For assertions it follows from the fact that if

con1 ⇔ con2 i.e. {con1} = {con2}

then

[con1].sta = tv iff [con2].sta =tv

In particular, this lemma implies that on the level of conditions (but not of boolean expressions of the programming layer!) we can apply all the lemmas of Sec. 9.3.2 that concern weak equivalence.

Lemma 9.4.2-4 *The replacement in a correct metaprogram of any boolean expression* vex *in an instruction by a boolean expression* vex1 *that is stronger defined (i.e., such that* vex ⊑ vex1*) does not violate the correctness of the metaprogram.*

If the source metaprogram is correct, then none of its boolean expressions generates an error, and wherever vex is defined vex1 is defined as well, and has the same value. In particular we may replace any boolean expression (and, of course, any conditions) by strongly equivalent ones.

9.4.3 Universal rules

First of our universal rules[82], that we shall call the *main rule,* bases on our earlier assumption about the general structure of metaprograms and is the following:

Rule 9.4.3-1 The basic rule

```
(1) pre prc                               : spp              post (de-con and in-con)
(2) pre (de-con and in-con)               : open procedures  post (de-con and op-con and in-con)
(3) pre (de-con and op-con and in-con)    : sin              post (de-con and op-con and si-con)
-----------------------------------------------------------------------------------------------
pre prc:
   spp ; open procedures ; sin
post (de-con and op-con and si-con)
```

In this rule:

- spp is a specified program preamble,
- de-con is a hereditary condition induced by declarations included in spp,
- in-con is a condition induced by instructions included in spp,
- op-con is a hereditary condition induced by **open procedures**,
- sin is a specified instruction,
- si-con is a condition induced by sin.

Although our rule is quite straightforward, we decided to show it, since it illustrates a way in which a final postcondition of a metaprogram is constructed incrementally. This rule bases on the observation from Sec. 9.3.3 that postconditions induced by declarations are immunizing in **Lingua-V**, and also on an obvious fact that condition in-con is resilient to **open procedures**. In the third step of our program development, in-con in the precondition is modified to si-con in the postcondition. This modification describes “the real effect” of the execution of our program.

Note now that whereas an incremental construction of a final postcondition is explicit in our rule, the process of building precondition is not. It is only implicit in the fact that all declaration-induced conditions — that are necessary to make our programs run cleanly — need some underivable preconditions to be induced. We have to make the latter primitive in our metaprogram, and, of course, they will be temporary. Each of them will cease to be satisfied when it is “consumed” by an associated declaration.

[82] Mathematically program-construction rules described in this and the following sections are just lemmas as in preceding sections. We call them “rules” for historical reasons, but must remember that they are not “assumed” as in Hoare’s or Dijkstra’s logic, by have to be proved (unless are obvious).

Our main rule is based on the following universal rule for sequential composition:

Rule 9.4.3-2 Sequential compositions

```
pre prc-1: spr-1 post poc-1
pre prc-2: spr-2 post poc-2
poc-1 ⇨ prc-2
------------------------------------------------
pre prc-1: spr-1;                  spr-2 post poc-2
pre prc-1: spr-1; asr poc-1 rsa; spr-2 post poc-2
pre prc-1: spr-1; asr prc-2  rsa; spr-2 post poc-2
```

Proof is immediate from Rule 8.7.1-1. Under the line we have a conjunction of metaconditions which means that our rule represent three single rules. The second and the third version will be used in *transformational programming* sketched in Sec. 9.4.6.6.

Rule 9.4.3-3 Strengthening preconditions

```
pre prc : spr post poc
prc-1 ⇨ prc
------------------------------------------------
pre prc-1 : spr post poc
```

Rule 9.4.3-4 Weakening postconditions

```
pre prc : spr post poc
poc ⇨ poc-1
------------------------------------------------
pre prc : spr post poc-1
```

Rule 9.4.3-5 Conjunction and disjunction of conditions

```
pre prc-1 : spr  post poc-1
pre prc-2 : spr  post poc-2
------------------------------------------------
pre (prc-1 and prc-2) : spr  post (poc-1 and poc-2)
pre (prc-1 or prc-2)   : spr  post (poc-1 or poc-2)
```

Rule 9.4.3-6 Propagation of resilient conditions

```
pre prc: spr post poc
con resilient to spr
------------------------------------------------
pre (prc and con) : spr post (poc and con)
```

The proofs of these rules follow immediate from the rules 8.7.1-3, 8.7.1-4, 8.7.1-5 and 8.7.1-6 respectively.

9.4.4 Rules for metadeclarations

There are four categories of atomic declarations (Sec. 6.7.1) to be considered from the perspective of program-construction rules:

- declarations of variables,
- enrichments of covering relations,
- declarations of classes,
- global openings of procedures.

In all these cases postconditions of corresponding metadeclarations are built in an incremental way. We shall discuss them in the subsequent sections.

9.4.4.1 Variable declarations

The rule for variable declarations is nuclear and is the following:

Rule 9.4.4-1 Variable declaration

```
pre (ide is free) and (tex is type)
   let ide be tex with yex tel
post var ide is tex with yex
```

The proof is obvious. The rule for class attribute declaration is analogous, but belongs to a different category since attribute declarations are executed as components of class declarations.

9.4.4.2 Enrichment of a covering relation

An enrichment of a current covering relations add new pairs of types to this relations and modifies cov-expression accordingly. This leads us to the following rule:

Rule 9.4.4-2 Enrichment of a covering relation

```
pre consistent(tex1 , tex2) and (coe is current):
   enrich-cov(tex1, tex2 )
post ((tex1, tex2) ; coe ) is current
```

The situation with condition coe **is current** is similar to that of ide **is free**. It is not derivable and va-hereditary (Sec. 9.3.3). In turn the derivability of **consistent**(tex1 , tex2) must be insured during the process of program derivation, depending on what tex1 and tex2 are.

9.4.4.3 Class declarations

A general scheme of a class metadeclaration is the following:

```
pre prc:
   class ide parent cli with ctr-1; … ; ctr-k ssalc          (9.4.4-1)
post poc
```

where ctr-i's are atomic class transformers and cli is a class indicator which may be of one of two following forms:

```
cli : Identifier
cli = empty-class
```

Our goal in the development of this metadeclaration consists in establishing:

- a precondition prc that guarantee a clean execution of our declaration,
- a postcondition poc that describes the effect of this declaration.

To realize this goal let's rewrite (9.4.4-1) to an equivalent form where class transformers are replaced by anchored class transformers:

```
pre prc :
   class ide parent cli with skip-ctr ssalc ;
   ctr-1 in ide ;
   …
   ctr-k in ide
post con
```

Given this form we can formulate a scheme of a rule analogous to the main rule in Sec. 9.4.2:

Rule 9.4.4-2 Class declaration

```
(1) pre prc                        : class ide parent cli with skip-ctr ssalc   post pa-poc
(2) pre pa-poc                     : ctr-1 in ide   post (pa-poc and cr-poc-1)
(3) pre (pa-poc and cr-poc-1) : ctr-2 in ide   post (pa-poc and cr-poc-1and cr-poc-2)
(4) ...
-----------------------------------------------------------------------------
pre prc:
   class ide parent cli with ctr-1; … ; ctr-k ssalc
post poc
```

The proof of this rule is immediate from Rule 9.4.2-1 for sequential composition. What remains to be done now, is to define rules for all categories of metadeclarations that may appear above the line.. First of them concerns a declaration of a funding class and is the following:

Rule 9.4.4-3 Declaration of a funding class

```
pre : (cl-ide is free) and (cli is class)
   class cl-ide parent cli with skip-ctr ssalc
post ide child of cli
```

Given this *initiation rule* we can proceed to the rules for anchored class transformers. To save the space (and the resilience of our readers!), we will show only selected examples of such rules. We start from the rule for adding an attribute. It is similar to Rule 9.4.4-1 for variable declaration.

Rule (9.4.4-4) Adding an abstract attribute

```
pre (at-ide is free) and (cl-ide is class) and (tex is type) :
   let at-ide be tex with yex as pst tel in cl-ide
post att at-ide is tex with yex in cl-ide as pst
```

Rule 9.4.4-5 Adding a type constant

```
pre (tc-ide is free) and (cl-ide is class) and (tex is type) :
   set tc-ide be tex tes in cl-ide
post tc-ide is tex
```

Rule 9.4.4-5 Adding an imperative pre-procedure declaration

```
pre (pr-ide is free) and (cl-ide is class)
   pr-ide (val my-fpc-v ref my-fpc-r) my-body in cl-ide;
post pre-proc pr-ide (val my-fpc-v ref my-fpc-r) my-body imperative in cl-ide
```

The postcondition of the resulting metadeclaration has been defined in Sec. 9.2.5. The soundness of this rule is evident from the definition of the applied transformer (Sec. 6.7.4.6). Note that all we need for a pre-procedure declaration to execute cleanly is that its hosting class cl-ide has been declared, and its name pr-ide is fresh. Rules for functional procedures and object constructors are, of course, analogous.

9.4.5 The opening of procedures

Our last rule associated with declarations concerns the global declaration **open procedures**. In this case we can't formulate one universal rule since the number of class declarations in a program, and the numbers of procedure declarations in each class are unlimited. All we can do, is to formulate the following scheme of a nuclear rule.

Rule 9.4.5-1 The opening of procedures

```
pre
   pre-proc pr-ide-11 (val fpc-v-11 ref fpc-r-11) body-11 imperative in cl-ide-1 and
   pre-proc pr-ide-12 (val fpc-v-12 ref fpc-r-12) body-12 imperative in cl-ide-1 and
   …
```

pre-proc pr-ide-21 **(val** fpc-v-21 **ref** fpc-r-21**)** body-21 **imperative in** cl-ide-2 **and**
pre-proc pr-ide-22 **(val** fpc-v-22 **ref** fpc-r-22**)** body-22 **imperative in** cl-ide-2 **and**
…
open procedures
post
cl-ide-1.pr-ide-11 **opened,**
cl-ide-1.pr-ide-12 **opened,**
…
cl-ide-2.pr-ide-21 **opened,**
cl-ide-2.pr-ide-22 **opened,**
…

It is to be emphasized that whereas in the precondition we have a conjunction of single conditions, the post-condition is one atomic condition that expresses a property of a tuple of procedures. Such a construction is necessary, since global declarations declare tuples of procedures in "one step". Therefore, our postcondition should express the fact that procedures assigned to procedure indicators in the current state are identical with procedures created from the corresponding pre-procedures.

9.4.6 Rules for metainstructions

9.4.6.1 Rules for composed instructions

Rule 9.4.6-1 Conditional branching if-then-else-fi

pre (prc **and** vex) : sin1 **post** poc
pre (prc **and not** vex) : sin2 **post** poc
prc ⇨ (vex **or** (**not** vex))
---
pre prc : **if** vex **then** sin1 **else** sin2 **fi post** poc

Here the two-sided vertical arrow represents two implications: top-to-bottom and a bottom-to-top. The metaimplication above the line guarantees that whenever the precondition is satisfied, the evaluation of boolean expression vec terminates, and yields a boolean value rather than an error. Note that in a two-valued logic this metadeclaration would be a tautology, and therefore is omitted.

The second rule corresponds to a **while-do-od** loop, where vex is a boolean expression, and inv is a condition called an *invariant of the loop*:

Rule 9.4.6-2 Loop while-do-od

(1) **pre** (inv **and** vex) **:** sin **post** inv
(2) **limited replicability of** (**asr** vex **rsa ;** sin) **if** inv
(3) prc ⇨ inv
(4) inv ⇨ (vex **or** (**not** vex))
(5) inv **and** (**not** vex)) ⇨ poc
---
pre prc **: while** vex **do** sin **od post** poc

The metacondition used in (2) has been defined in Sec. 9.3.1. Proof follows directly from rule Rule 8.7.2-6.

9.4.6.2 Rules for assignment instructions

In the case of assignment instructions, instead of formulating a rule "ready to be used", we show a universal rule and sketch a way of using it in concrete situations. This rule has a tautological character and is the following:

Rule 9.4.6-1 @-tautology

pre sin **@** con

```
   sin
post con
```

The proof of this rule follows directly from the definition of the denotation of sin @ con in Sec. 9.2.7. The idea of using this rule consists in a replacement of the algorithmic precondition by a weakly equivalent one which is not algorithmic. To see, how it works consider as an example the following tautological metainstruction:

```
pre x := y+1 @ 2*x < 10
   x := y+1                                  (9.4.6.2-1)
post 2*x < 10
```

where we assume that the arithmetical operators +, * and < are integer operations. It is implicit in this rule that:

x   has been declared as an integer variable and (therefore) its value is an integer
y   analogously
x+1 does not generate an error
2*x analogously

Under this assumption we can easily prove the following weak equivalence (note that a strong equivalence does not hold):

```
x := y+1 @ 2*x < 10   ⇔  (x is integer) and 2*(y+1) < 10      (9.4.6.2-2)
```

Due to our assumptions about arithmetical operators the left-hand side of the equivalence implies that x and y are integer variables. Since on the right-hand side x does not appear in the inequality, we have to add an explicit claim about its type. For simplicity we do not consider the possibility of an overload, and we assume that the types of both variables are yokeless, i.e. that their yokes are TT.

By Rule **9.4.3**-3, the precondition of (9.4.6.2-1) may be replaced by the right-hand side of (9.4.6.2-2) which leads us to the following metainstruction, which is no more a tautology:

```
pre (x is integer) and 2*(y+1) < 10
   x := y+1
post 2*x < 10
```

This step completes the development of a correct metaprogram.

Now, let's apply Rule 9.4.6-1 in an object-oriented context. Consider the following initial tautological metainstruction:

```
pre x.q := y.p.r + 1  @  2*x.q < 10
   x.q := y.p.r + 1
post 2*x.q < 10
```

where x, y and y.p point to objects. Now, we can prove the following weak equivalence:

```
x.q := y.p.r + 1  @ 2*x.q < 10
 ⇔
(type of x.q accepts type of y.p.r+1) and 2*(y.p.r+1) < 10
```

Note that it is implicit in (i.e. is metaimplied by) the right-hand side of this equivalence that x, y and y.p point to objects, and that all involved expressions evaluate cleanly. Basing on this equivalence we can claim the correctness of the following metainstruction:

```
pre (type of x.q accepts type of y.p.r) and 2*(y.p.r+1) < 10
   x.q := y.p.r + 1
post 2*x.q < 10
```

9.4.6.3 Rules for imperative procedure calls

Construction rules formulated so far might be seen as tools for handling the following programming tasks:

A. given a postcondition poc of a future metaprogram,
B. create a specified program spr, and a precondition prc such that the following metaprogram is correct:

   pre prc: spr **post** poc

In the case of procedure calls the task is different. This time,

C. given a postcondition poc-call of a future procedure call,
D. create a procedure declaration

   proc myProc **(** **val** fpa-v **ref** fpa-r **)** **begin** my-body **end**, (9.4.6.3-1)

   and a precondition prc-call such that the following metainstruction is correct:

   pre prc-call **:**
   call MyClass.myProc(**val** apa-v **ref** apa-r) (9.4.6.3-2)
   post poc-call.

Of course, in the realization of this task, the major subtask and challenge consists in developing a correct metaprogram

pre prc-body**:**
my-body
post poc-body.

that includes the body of our future procedure. In the realization of D. we have to develop the following elements of the future declaration and call:

1. a procedure body my-body,
2. conditions prc-body and poc-body, such that
   pre prc-body**:** my-body **post** poc-body
   is correct,
3. two lists of formal parameters fpa-v and fpa-r,
4. two lists of actual parameters apa-v and apa-r,
5. a precondition of the call pre-call such that (9.4.6.3-2) is correct.

We shall try to figure out what relationships between the expected elements in 1. − 5. should hold to make (9.4.6.3-2) satisfied.

In the first place the precondition of the call must guarantee that procedure myProc has been declared in MyClass, and that it has been opened. This prerequisite may be expressed by two following metaimplication:

prc-call ⇨ myProc (**val** fpa-v, **ref** fpa-r) **begin** my-body **end imperative in** MyClass
prc-call ⇨ **procedure** MyClass.myProc **opened**

Both conditions in these metaimplications were defined in see Sec. 9.2.5. It is implicit in the first one that MyClass has been declared.

The third fact that prc-call must guarantee is that the passing of actual parameters to formal parameters will execute cleanly, and that the resulting state will satisfy prc-body. To express this fact we shall use algorithmic condition (9.2.5-1) defined in Sec. 9.2.5, and request the following metaimplication:

prc-call ⇨ **pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** MyClass **@** prc-body

There is, however, one technical problem hidden in this request. When function pass-actual defined in Sec. 6.6.3.4 is generating a local-initial state, this state is getting a declaration-time environment dt-env and a call-time store (cf. Sec. 6.6.3.2). In turn, when we evaluate prc-call we are dealing with a call-time environment ct-env, rather than declaration-time environment dt-env. In this place we should recall that pass-

actual "uses" the environment exclusively to compute the types of formal parameters, and the types declared in ct-env are the same as types declared in dt-env, which, in turn, is a conclusion of our assumption (see Sec. 6.3) that in programs there are no declarations that would follow the opening of procedures. Consequently, our metaimplication describes adequately our expectations.

The third, and the last fact that we have to guarantee, is that the satisfaction of postcondition poc-body in a local terminal state lt-sta will guarantee:

a. that the return of references of formal parameters to actual parameters will be executed without an error message,
b. that after the return of parameters poc-call will be satisfied in the global terminal state.

The requirement b. may be expressed by the following metaimplication:

poc-body[fpa-r/apa-r] ⇨ poc-call

where poc-body[fpa-r/apa-r] denotes poc-body, where each formal reference-parameter has been replaced by the corresponding actual parameter. Note that for every formal parameter there is exactly one actual parameter (although not necessarily vice versa).

To express requirement a. we have to cope with another technical problem such that the references of actual parameters have to accept the values of corresponding formal parameters in the context of the declaration-time covering relation dt-cov, i.e. in the declaration-time environment. By the assumption that all declarations in our programs precede the openings of procedures (Sec. 6.3), and therefore also procedure calls, we can claim that this relation equals the call-time relation ct-cov, but still we have to express a property of a local-terminal state in referring to the cov-relation of a "remote" state (see. Fig. 6.6-3). Note in this place that local-terminal relation may be different from (global) call-time relation since an extension of this relation might have taken place in the body of our procedure.

What we have to do in this situation, is to "recall" ct-cov, "remembered" in a cov-expression coe that was adequate at the entrance to the call, i.e., that satisfies metaimplication

prc-call ⇨ coe **is adequate**

In practice, prc-call will „conjunctively include" condition coe **is adequate**. Given the call-time coe we can request the metaimplication

poc-body ⇨ fpa-r **accepts** apa-r **in** coe

which means that prc-body assures a clean execution of passing reference parameters. Summing up our considerations, we may claim the soundness of the following rule:

Rule 9.4.6.3-1 A call of an imperative procedure

(1) prc-call ⇨ myProc (**val** fpa-v **ref** fpa-r) my-body **imperative in** MyClass
(2) prc-call ⇨ (**pass actual val** apa-v **ref** apa-r **to formal val** fpa-v **ref** fpa-r **with** MyClass) **@** prc-body
(3) prc-call ⇨ **procedure** MyClass**.**myProc **is opened**
(4) prc-call ⇨ coe **is current**
(5) prc-body ⇨ my-body **@** poc-body   i.e. **pre** prc-body **:** my-body **post** poc-body
(6) poc-body ⇨ fpa-r **accepts** apa-r **in** coe
(7) poc-body[fpa-r/apa-r] ⇨ poc-call

---

pre prc-call **:**
  call MyClass.myProc (**val** apa-v **ref** apa-r)
post poc-call

Let us comment this rule.

The precondition of the call guarantees that:

(1) A pre-procedure named myProc has been declared in MyClass and the denotation of its body is identical with [my-body] (cf. Sec. 9.4.6.3). Note that this condition does not say (!) that the body of our procedure is my-body.

(2) The process of passing actual parameters to formal parameters terminates cleanly, and the output state satisfies the precondition of the body.

(3) Procedure MyClass.myProc has been opened, which practically means that the input state is a "follower" of an output state of declaration **open procedures**. Note that in syntactically correct programs metaimplication (3) is a consequence of (1) due to the rule (cf. Sec. 6.3) that no declarations follow **open procedures**. However, we decided to put (3) into our rule, to make it context-independent. In other words, we attempt to express all assumptions necessary for the correctness of our procedure call in terms of the properties of its input states.

(4) Covering expression coe describes adequately the covering relation of the call-time state of the procedure.

The precondition of the body guarantees that:

(5) Every program whose denotation is [my-body] — hence, in particular, the body of our procedure — when starting its execution with prc-body satisfied, terminates cleanly, and its output state satisfies poc-body.

The postcondition of the body guarantees that:

(6) The process of returning formal parameters to actual parameters terminates cleanly.

(7) After the return of the references of formal reference-parameters to actual reference-parameters the postcondition of the call will be satisfied.

9.4.6.4 The case of recursive imperative procedures

In Sec. 9.4.6.3 we have introduced a sound construction rule which we can use in the process of building a procedure declaration with expected properties. In this process we have to "invent" such components of a future procedure declaration that the future call of this procedure will be correct for a given postcondition, and an "invented" precondition. Among seven metaimplications that we have to prove in order to ensure the correctness of our call, the implication (5) states that the body of our procedure is correct. In practice, we shall not prove (5) after having developed my-body, but we shall develop a correct metaprogram of the form:

```
pre prc-body
   my-body                                                    (9.4.6.4-1)
post poc-body
```

Let's try to figure out now, what happens, if my-body includes a (recursive) call of the future procedure? In this case our rule is still adequate, but we can't establish the correctness of (9.4.6.4-1) without assuming the correctness of the future call:

```
pre prc-call :
   call MyClass.myProc (val apa-v ref apa-r)                  (9.4.6.4-2)
post poc-call
```

In other words, in the case of recursion, we can't first build (9.4.6.4-1) and then claim (9.4.6.4-2), but we have to develop/prove them in a certain sense "in parallel".

Further on, our procedure may call itself more than once in its body, and not necessarily directly, but also via other procedures. There is, therefore, a potentially infinite number of different mutual-recursion configurations that lead to an infinite number of corresponding construction rules. This situation may be compared to the case of iterative programs with goto's considered in Sec. 8.3. However, whereas in the latter case a way out of the "labyrinth" of a variety of different programming structures was offered by structured programming, an analogous solution for recursive procedures seems doubtful. In the case of recursion, we probably

have to treat each recursive structure separately using general rules described in Sec. 8.7.2. We will not delve deeper into this problem, leaving it for further research. Instead, we shall analyze one simple example.

Let power be the name of a (future) recursive procedure to be declared in MyClass[83], and let's assume that our task consists of making the following call correct (we use some obvious colloquializations):

```
pre prc-call
    call MyClass.power(val a,b ref c)                                        (9.4.6.4-3)
post var a,b,c are integer with value ≥ 0 and c=a^b
```

where prc-call is to be found. As a "candidate declaration" of our procedure declaration let's take:

```
proc MyClass.power(val m, n integer ref k integer)      k = m^n
    begin
        if n = 0
            then k := 1                                                      (9.4.6.4-4)
            else n := n-1 ; call MyClass.power(val m, n ref k); k:= k*m
        fi
    end
```

and as a candidate for the precondition of the call let's take:

```
prc-call =  power (val m, n integer ref k integer) my-body imperative in MyClass and
            var a,b,c are integer with a,b,c ≥ 0
```

where my-body is implicit in (9.4.6.4-4). To prove the correctness of (9.4.6.4-3) we shall use Rule 8.7.2-4 (Sec. 8.7.2). Let:

```
A = { prc-call }
B = { var a,b,c are integer with a,b,c ≥ 0 and c=a^b }

H = [ asr b > 0 rsa ] [ b := b-1 ]
T = [ c:= c*a ]
E = [ asr b = 0 rsa ] [ c := 1 ]
F = [ call MyClass.power(val a,b ref c) ]
```

To be able to claim (9.4.6.4-3) have to prove the following statements:

(1) $(\forall Q)$ ($AQ \subseteq B$ *implies* $A(HQT) \subseteq B$)
(2) $AE \subseteq B$
(3) $A \subseteq FS$

where Q denotes a denotation of an imperative program-component. We leave the details of this proof to the reader.

9.4.6.5 The case of functional procedures

Analogously to imperative procedures, correctness statements about functional procedures describe the properties of their calls. In this case, however, the result of a call is not a state, whose properties may be described by a condition, but a value. Consider the following (anchored) declaration of a functional pre-procedure:

```
fun funPower(val k, m, n) functional in MyClass
    begin
        let k be integer with value ≥ 0 tel
        call MyClass.power(val m, n ref k)                                   (9.4.6.5-1)
        return 3*k+1
    end
```

[83] In this case we typeset power and MyClass in green Arial Narrow, since contrary to the case of Sec.9.4.6.3, now we are talking about a concrete procedure, rather than about a pattern of a procedure.

where MyClass.power is the procedure analyzed in Sec. 9.4.6.4. A possible correcness statemet describing a property of of the call may be the following:

(funPower **(val** m, n **ref** k**) begin** body **return** integer **end functional in** MyClass **) and (** a, b ≥ 0 **) :**
⇨
call funPower(a, b) = 3*(a^b)+1

where body is implicit in (9.4.6.5-1). This statement expresses the relationship between the input values of actual parameters a, and b, and the value exported by the call. Note that we may write our statement in a standard form with pre- and postcondition as:

pre funPower **(val** m, n **ref** k**) begin** body **return** integer **end functional in** MyClass **and** a, b ≥ 0:
skip-i
post call funPower(a, b) = 3*(a^b)+1

It is implicit in the postcondition that the call evaluates cleanly.

We shall not go into a discussion of building correct functional procedures, leaving it to future research. In the same "spirit" we abandon a discussion of the construction of correct object constructors.

9.4.6.6 Jaco de Bakker paradox in Hoare's logic

As was noticed by Jaco de Bakker (p. 108, Sec. 4 in [8]) and later commented by K. Apt in [4], on the ground of Hoare's logic one can prove the formula:

pre true **:** a[a[2]] := 1 **post** a[a[2]] = 1

which for same arrays a is not true. Indeed, if

a = [2,2].

then

a[2] = 2

hence the execution of the assignment

a[a[2]] := 1

means the execution of

a[2] := 1

which means that the new array is a = [2,1], and therefore a[a[2]] = a[1] = 2.

Let us observe, however, that Hoare's problem results neither from having arrays in a language nor from the admission of expressions like a[a[2]] but from an implicit assumption that whenever such an expression appears on the left-hand-side of an assignment, it should be treated as a variable. As a matter of fact, for many years, programmers used to talk about "subscripted variables" (in Algol 60 [7]) or "indexed variables" (in Pascal [62]).

De Bakker's problem with Hoare's logic lies in an imperfect understanding of the meaning (the semantics) of array variables[84]. In our language de Bakker's paradox does not appear since the instruction of the form:

a.(a.2) := 1

would be syntactically incorrect. In that place, we write

a := **change-in-arr** a **at** a.2 **by** 1 **ee**

or colloquially

[84] In the denotational model described by M. Gordon in [59] array-variables or indexed-variables are admitted on the cost of a rather substantial complication of the model by distinguishing between left-values of expressions (locations) and right-values of expressions (values).

```
a := change-in-arr a by a.2 := 1 ee
```

Now, on the ground of constructions rules of Sec. 9.4 we can easily derive the following correct metaprogram:

```
pre a is arr-type number and a.1=2 and a.2=2
   a := change-in-arr a by a.2 := 1 ee
post a.1=2 and a.2=1
```

9.5 Transformational programming

9.5.1 First example

In the previous section, we were dealing with rules allowing to build correct metaprograms out of correct components. That was a situation analogous to an assembly line of, e.g., automobiles. In the present section, we shall consider rules to be used in metaprogram transformations, when we want to change or to optimize program functionality. In the examples that follow, we shall use some of the rules introduced earlier as well as some others that we are going to formalize in Sec. 9.5.3. Let us start with an example of two obviously correct metaprograms, where we assume that nnint is a predefined type of non-negative integers. Since all our variables will be yokeless, we shall skip for simplicity the phrase "**with TT**".

```
pre x,n is nnint :
   x := 0;
   while (x+1)² ≤ n
   do
      x := x+1
   od
post x = isrt(n)
```

```
pre x,n,m is nnint
   x := 0;
   while (x+1)*m ≤ n
   do
      x := x+1
   od
post x = iqt(n,m)
```

The first program computes an integer square root denoted by isrt(n), the other — an integer quotient denoted by iqt(n). Each of these metaprograms is searching number-by-number through the set of nonnegative integers in seeking the expected result. Returning to our automotive metaphor, we may say that both metaprograms are driven by the same while-engine:

```
P1:  pre x,k is nnint:
        x := 0;
        while x+1 ≤ k
        do
           x := x+1
        od
     post x = k
```

We can use this universal engine to drive two different "appliances": an integer square root, or an integer quotient. In each of these cases, we change the functionality of a program but preserve its correctness. Let us show a simple universal method that can justify the correctness of the resulting metaprogram.

First observe that the correctness of P1 implies the correctness of P2, where we introduce an assertion block (Sec. 9.2.6) including **while** instruction, and where k has been replaced by isrt(n):

```
P2:  pre x,n is nnint :
        x := 0;
        asr x,n is nnint in
           while x+1 ≤ isrt(n)
           do
              x := x+1
           od
        rsa
```

```
post x = isrt(n)
```

So far, our metaprogram looks a bit pointless since it uses isrt(n) to compute it. We shall, therefore, eliminate that expression from the programming layer basing on a strong equivalence[85]:

$x+1 \le isrt(n) \equiv (x+1)^2 \le n$ **whenever** x,n **is** nnint

and applying Lemma 9.4.2-4 (Sec. 9.4.2), which allows replacing a boolean expression by a strongly equivalent one. In our case, this equivalence holds only in the context specified by the **whenever** clause, and this context is assured within the our assertion block.

As a result of the described transformation, we end up with a final metaprogram P3 where the assertion (now not necessary) has been removed.

```
P3:  pre x,n is nnint :
       x := 0;
       while (x+1)² ≤ n
         do
           x := x+1
         od
     post x = isrt(n)
```

The instruction of the derived metaprogram does not refer to isrt(n) anymore, and therefore may be said to be "more practical" than P2.

Still, our program is very slow. If we want to speed it up, we have to install a "faster engine" to drive it. Let us start from the construction of a universal searching engine for "target integers" in a logarithmic time.

Let po2.k denote a condition which is satisfied if k is a nonnegative power of 2, i.e., if there exists a nonnegative m such that:

$k = 2^m$

Let mag.k (the magnitude of k) denote a function with values in the set of powers of 2 such that

$mag.k \le k < 2*mag.k$

For instance, $mag.11 = 2^3$ since

$2^3 \le 11 < 2^4$

Now, it is easy to prove the total correctness of the two following metaprograms:

```
Q1:  pre x,k,z is nnint :
       z := 1;
       asr x,k,z is nnint and po2.z in
         while z ≤ 2*mag.k do z:=z*2 od
       rsa
     post x,k,z is nnint and z = 2*mag.k
```

and

```
Q2:  pre x,k,z is nnint and z = 2*mag.k:
       x := 0;
       while z > 1
         do
           z := z/2;
           if x+z ≤ k then x:=x+z else skip-i fi
         od
     post x = k and z = 1
```

[85] This equivalence may be formally proved on the ground of the following definition: isrt(n) is the unique integer k such that $k^2 \le n < (k+1)^2$.

The first metaprogram computes the successive powers of 2 until it reaches 2*mag.k, and the second returns from 2*mag.k to 1 through successive powers 2^m and on its way summarises these powers of 2 that correspond to 1 in the binary representations of k. For instance, since

11 = 0*16 + 1*8 + 0*4 + 1*2 + 1*1

the second metaprogram, while given 2*mag.11 = 16, will perform the following summation

8 + 2 + 1 = 11.

In this way, the target value of k is reconstructed in logarithmic time, compared to a linear time of metaprogram P3. Now observe that the following metacondition is true:

z ≤ mag.k ≡ z ≤ k **whenever** x,n,z **is** nnint **and** po2.z

Due to this equivalence, we can replace the boolean expression in **while** of the first metaprogram by the strongly equivalent expression z ≤ k. If we join both metaprograms on the ground of Rule 9.4.2-5, we get our target metaprogram that finds the value of k in logarithmic time. In the same step, we move the initialization of x at the beginning of the metaprogram.

```
Q3: pre z, x, k is nnint :
        z := 1;
        x := 0;
        asr x,k,z is nnint and po2.z in
            while z ≤ k do z:=2*z od;
            while z > 1
                do
                    z := z/2;
                    if x+z ≤ k then x:=x+z fi
                od
        rsa
    post x = k and z = 1
```

Here and in the sequel

if vex **then** ins **fi**

means

if vex **then** ins **else skip-i fi**

If in Q3 we replace the expression k by the expression isrt(n), then we have a program that computes isrt(n) but refers to it. We eliminate isrt(n) by using two strong conditional equivalences:

z ≤ isrt(n) ≡ $z^2 \le n$ **whenever** z, n **is** nnint
x+z ≤ isrt(n) ≡ $(x+z)^2 \le n$ **whenever** z, x, n **is** nnint

In this way we get

```
Q4: pre z, x, n is nnint:
        z := 1;
        x := 0;
        asr x,k,z is nnint and po2.z in
            while z² ≤ n do z:=2*z od ;
            while z > 1
                do
                    z := z/2;
                    if (x+z)² ≤ n then x:=x+z fi
                od
        rsa
    post x = isrt(n) and z = 1
```

Now we shall time-optimize our program by restricting the number of performed operations. Let us start from the observation that in each run of the first loop, the program recalculates the value of z^2, which is not optimal. To speed up Q4 we introduce a new variable q, and we enrich our program in such a way that the condition $q=z^2$ is always satisfied. Such a q will be called a *register identifier* and z^2 — a *register expression*. This technique is discussed in details in Sec. 9.5.3.

```
Q5: pre z, x, n, q is nnint:
      z := 1;
      x := 0;
      q := 1;
      asr z, x, n is nnint and po2.z and q = z² rsa
        while q ≤ n
          do
            off z:=2*z; q:=4*q ffo
          od
        while z > 1
          do
            off z:=z/2; q:=q/4 ffo
            if x²+2*x*z+q ≤ n then x:=x+z fi
          od
      rsa
    post x=isrt(n) and z = 1 and q=z²
```

Note that the double-use of **off-ffo** is necessary since each time when the first assignment destroys the satisfaction of $q=z^2$, the second recovers it. For better readability of our program we do not "quote" the assertion in the off-ffo instruction assuming that it is defined by the context. Now we proceed to further transformations:

1. we use the equivalence $z>1 \equiv q>1$ **whenever** ($z>0$ **and** $q=z^2$) to modify boolean expression in the second loop,
2. we introduce two new variables y and p with the conditions $y = n\text{-}x^2$ and $p = x*z$,
3. we use the equivalence $x^2 + 2*x*z + q \le n \equiv 2*p+q \le y$ **whenever** ($y=n\text{-}x^2$ and $p=x*z$)

Using the corresponding transformations, we get the following program

```
Q6: pre z, x, n, q, y, p is nnint:
      z := 1;
      x := 0;
      q := 1;
      asr z, x, n is nnint and q = z2 in
        while q ≤ n
          do
            off z:=2*z; q:=4*q ffo
          od
        y := n;
        p := 0;
        asr y=n-x2 and p = x*z in
          while q > 1
            do
              off z:=z/2; q:=q/4; p:=p/2; ffo
              if 2*p+q ≤ y then x:=x+z; p:=p+q; y:=y-2p-q fi
            od
        rsa
      rsa
    post x=isrt(n) and z=1 and q=z² and y=n-x² and p=x*z
```

Contrary to the former introduction of a new variable which was clearly justified, now it not quite clear why p and y have been introduced. The answer follows from a well-known truth that in programming, like in playing chase, we sometimes have to predict a few moves in advance. These moves are shown a little later.

In the next transformation, we prepare our metaprogram for the removal of variable z. For that sake, we perform the following changes:

1. we apply the equivalence $q=z^2$ ⇔ isrt(q)=z **whenever** z > 0 to change the assertion,
2. we use the condition isrt(q) = z to replace z by isrt(q) everywhere except the left-hand side of the assignment,
3. we make obvious changes based on the equality z=1 .

The resulting metaprogram is the following:

```
Q7: pre z, x, n, q, y, p is nnint:
      z := 1;
      x := 0;
      q := 1;
      asr z, x, n is nnint and isrt(q)=z in
        while q ≤ n
          do
            off z:=2*isrt(q); q:=4*q ffo
          od
        y := n;
        p := 0;
        asr y = n-x2 and p = x*isrt(q) in
          while q > 1
            do
              off z:=isrt(q)/2; q:=q/4; p:=p/2 ffo
              if 2*p+q ≤ y then x:=x+isrt(q); p:=p+q; y:=y-2p-q fi
            od
        rsa
      rsa
    post x=isrt(n) and z=1 and q=1 and p=x and y=n-x2
```

Now observe that in Q7 the variable z does not appear neither in boolean expressions nor on the right-hand sides of assignment that do not change z. Since we do not care about the terminal value of z, we can remove that variable from our metaprogram together with the corresponding assignment (general rule will be described in Sec. 9.5.1). In this way we get:

```
Q8: pre x, n, q, y, p is nnint :
      q := 1;
      x := 0;
      asr x, n is nnint in
        while q ≤ n
          do
            q:=4*q
          od
        y := n;
        p := 0;
        asr y = n-x2 and p = x*isrt(q) in
          while q > 1
            do
              off q:=q/4; p:=p/2 ffo
              if 2*p+q≤y then x:=x+isrt(q); p:=p+q; y:=y-2p-q fi
            od
        rsa
```

```
        rsa
    post x=isrt(n) and q=1 and p=x and y=n-x2
```

Now we use the equivalence

```
x=isrt(n) ≡ p=isrt(n) whenever p=x
```

to modify the postcondition which makes variable x not necessary anymore. Therefore, we can remove it with all expressions, and assertions, where it appears.

```
Q9: pre n, q, y, p is nnint:
        q := 1;
        while q ≤ n do q:=4*q od
            y := n;
            p := 0;
            while q > 1
                do
                    if 2*p+q≤y then p:=p+q; y:=y-2p-q fi
                od
    post p=isrt(n) and q=1
```

In the last step we replace the instruction

```
p:=p/2; if 2*p+q≤y then p:=p+q; y:=y-2p-q else x:=x fi
```

by an equivalent instruction

```
if p+q≤y then p:=p/2+q; y:=y-p-q else p:=p/2 fi
```

As a result, we get the final version of our metaprogram:

```
Q10: pre n, q, y, p is nnint :
        q := 1;
        while q ≤ n do q:=4*q od
        y := n;
        p := 0;
        while q > 1
            do
                q:=q/4;
                if p+q≤y then p:=p/2+q; y:=y-p-q else p:=p/2 fi
            od
    post p = isrt(n)
```

This program was written by a well-known Norwegian computer-scientist Ole-Johan Dahl in 1970 to be applied in a microprogrammed arithmetical unit of a computer. It is very time-efficient since in a binary arithmetic the multiplications and divisions by 2 or 4, correspond to simple shifts left or right respectively of binary words. And except shifts it uses only addition and subtraction which are also time inexpensive. In the days when microprocessors were not very fast such optimization was worth the effort.

We do not know in what way Dahl has built this program but we may suppose that he performed an optimisation similar to ours, although without formalised rules.

Our example shows a certain specific approach to developing some programs with while-loops by building a program in three steps:

1. writing a program-engine that searches through a specific space of data,
2. installing an appliance on that engine which implements the expected functionality,
3. optimizing the program.

As we are going to see in Sec. 9.5.1, our technique may also be used in changing the types of data elaborated by a program.

9.5.2 Changing the types of data

The technique of register identifiers may be also used in the replacement of one data-type by another one. In this section we show how to transform metaprogram Q10 from Sec. 9.5.1 into a metaprogram that operates on binary representations of positive integers. Let

bin : Binary = {(0)} | {(1)} © {(0), (1)}^{c*}

be the set of binary representations of integers called *binary words*, and let

int : NnInt = {0, 1, 2,…}

be the set of non-negative integers. We shall use the following functions and relations defined on binary words:

```
sl : Binary ⟼ Binary                                        shift left
sl.bin =
   bin = (0)   ➔ (0)
   true        ➔ bin © (0)

sr : Binary ⟼ Binary                                        shift right
sr.bin =
   bin = (0)   ➔ (0)
   true        ➔ pop.bin

+ : Binary ⟼ Binary                                         addition
− : Binary ⟼ Binary                                         subtraction
< : Binary ⟼ {tt, ff}                                       less
≤ : Binary ⟼ {tt, ff}                                       less or equal
```

The addition and the subtraction of binary words are denoted by the same symbols as for numbers and we assume that they are defined in such a way that the equations (5) and (6) below are satisfied. The orderings are lexicographic and again correspond to their numeric counterparts.

```
b2n : Binary  ⟼ NnInt          binary to number; conversion function
n2b : NnInt   ⟼ Binary         number to binary; conversion function
```

All these functions and relations are defined in such a way that they satisfy the following equations:

```
(1) b2n.(n2b.lic)        = int
(2) n2b.(b2n.bin)        = bin
(3) n2b.(int*2)          = sl.(n2b.int)
(4) n2b.(int/2)          = sr.(n2b.int)        where „/” denotes the integer part of division
(5) n2b.(int1 + int2)    = n2b.int1 + n2b.int2
(6) n2b.(int1 − int2)    = n2b.int1 − n2b.int2
(7) n2b.int1 < n2b.int2  iff   int1 < int2
(8) n2b.int1 ≤ n2b.int2  iff   int1 ≤ int2
```

Now, we transform metaprogram Q10 by introducing to it three new variables and three corresponding register-conditions:

```
Q = n2b(q)
Y = n2b(y)
P = n2b(p)
```

We assume that a type binary has been defined in our language. We introduce the assertions into Q10 and we shift all initialisations to the beginning of our new metaprogram:

```
Q11:   pre n, q, y, p is nnint and Q, Y, P is binary and n ≥ 1
          q := 1; Q := (1);
          y := n; Y := n2b(n);
          p := 0; P := (0);
```

```
          asr Q = n2b(q) and Y = n2b(y) and P = n2b(p) in
          while q ≤ n
            do
              off q:=4*q ; Q = sl(sl(Q)) ffo
            od
          while q > 1
            do
              off q:=q/4; p:=p/2;
              Q:=sr(sr(Q)); P:=sr(P); ffo
              if p+q≤y
                then off p:=p/2+q; y:=y-2p-q; P:=sr(P)+Q; Y:=Y-sl(P)-Q ffo
                else off p:=p/2; P:=sr(P) ffo
              fi
            od
          rsa
        post p = isrt(n) and q = 1
```

Now we use four conditional equivalences in order to replace boolean numeric expressions by boolean binary ones:

```
q ≤ n      ≡  Q ≤ n2b(n)        whenever Q=n2b(q)
q > 1      ≡  (1) < Q           whenever Q=n2b(q)
p+q ≤ y    ≡  P+Q ≤ Y           whenever Q=n2b(q) and Y=n2b(y) and P=n2b(p)
p=isrt(n)  ≡  P=n2b(isrt(n))    whenever P=isrt(p)
```

Next we remove from our metaprogram all numeric variables except n with the corresponding assignments and the assertion block. Since this block reaches the end of the metaprogram, we can modify the postcondition in an appropriate way.

```
Q12: pre n ≥ 1 and Q, Y, P is binary
        Q := (1);
        Y := n2b(n);
        P := (0);
        while Q ≤ N do Q = sl(sl(Q)) od;
        while (1) < Q
          do
            Q:=sr(sr(Q)); P:=sr(P)
            if P+Q≤Y
              then P:=sr(P)+Q; Y:=Y-sl(P)-Q
              else P:=sr(P)
            fi
          od
     post P = n2b(isrt(n)) and Q = (1)
```

9.5.3 Adding a register identifier

This section is devoted to the transformation of metaprograms by adding to them a new identifier ide-r that satisfies an assertion of the form:

ide-r = vex-r.

Such transformations ware applied in Sec. 9.5.1 in passing from Q4 to Q5 and in Sec. 9.5.2 in passing from Q10 to Q11.

An identifier ide-r that satisfies the condition ide-r = vex-r in a certain range is called a *register-identifier* or just a *register;* the expression vex-r is called a *register-expression* and the condition ide-r = vex-r — a *register-condition.*

Let us start from an obvious generalization of the meaning of @ (Sec. 9.2.2) which now will compose instructions not only with conditions but also with value expressions:

[sin @ vex] = [sin] ● Sde.[vex]

Let's consider a metaprogram that we assume to be correct:

```
P:   pre prc
        sin-h;                                  head (possibly trivial)
        asr con rsa ;
        asr con in ins ; rsa
        sin-t                                   tail (possibly trivial)
     post poc
```

Let ide-r be an identifier which does not appear in P, and let vex-r be a value expression such that

pre con : ide-r := vex-r **post TT**

which simply means that con guarantees the execution of ide-r := vex-r without an error or looping. Under these assumptions a transformation that enriches P by introducing ide-r with a register-condition

ide-r = vex-r

yields a metaprogram:

```
Q:   pre prc and ide-r is tex
        sin-h ;
        ide-r := vex-r ;
        asr con and ide-r = vex-r in $(sin, ide-r = vex-r) rsa
        sin-t
     post poc
```

where $(sin, ide-r = vex-r) denotes such an enrichment of sin which makes Q correct, provided that P was correct. The assertion **asr** con **rsa** has been dropped from Q (although we could have left it there), since it only served to guarantee, that in its context the value of vex-r was defined.

The syntactic operation $ is defined by structural induction, wrt the structure of sin. Let us start from sin which is an assignment

ide := vex

where ide is different from ide-r, since we have assumed that ide-r does not appear in P.

If ide does not appear in vex-r, then the execution of this assignment does not cause any change in the value of vex-r, and therefore we do not need to add any actualization.

If, however, this is not the case, then directly after ide:=vex, we have to add an assignment which recovers the satisfaction of the condition ide-r = vex-r. In such a case

$(ide := vex , ide-r = vex-r) = **off** ide := vex; ide-r := vex-r **ffo**

where equality sign '=' denotes the equality of syntactic elements. An off-clause is necessary here since ide appears in vex-r. Consequently, the alteration of the value of ide may cause the alteration of the value of vex-r and the falsification of our condition. In the case of the transformation of Q4 to Q5 with a register condition $q=z^2$ this has led to the enrichment of

asr $q=z^2$ **rsa** ; z:=2*z

into:

asr $q=z^2$ **rsa** ; off z:=2*z ; q:=z2 ffo

The assertion has been left in the resulting instruction since we shall need it a little later. Now, our instruction may be changed into an equivalent one (note the inverse order of assignments):

asr $q=z^2$ **rsa** ; **off** q:=((z:=2*z) @ z^2) ; z:=2*z **ffo**

In this instruction, we can eliminate @, by transforming the expression (z:=2*z) @ z^2 to a standard form:

asr q=z^2 **rsa** ; **off** q:=4*z^2 ; z:=2*z **ffo**

Now, since the assertion q=z^2 holds "just before" the assignment q:=z^2, we can replace our instruction by:

asr q=z^2 **rsa** ; **off** q:=4*q ; z:=2*z **ffo**

which makes the modification of q independent of z, and therefore — in our example — allows for the elimination of z from the metaprogram. In the general case, these transformations are as follows. First the instruction

off ide:=vex ; ide-r:=vex-r **ffo**

is replaced by an equivalent one

off ide-r := ((ide := vex) @ vex-r) ; ide := vex **ffo**

Further on, the expression ((ide := vex) @ vex-r) is transformed to a standard form, and then we try to change it is such a way that the identifier ide can be eliminated due to the register-condition ide-r=vex-r. This action completes the transformation.

The second "atomic" case to be investigated is a procedure call:

call ide(**val** acp-v **ref** acp-r)

Let us assume that our procedure call appears in a program in the same context as the assignment in the former case. We again have two subcases to be considered.

If none of the actual referential parameters appears in vex-r, then we keep the instruction unchanged. In the opposite case, we replace it with the instruction

off call ide (**ref** acp-r **val** acp-v); ide-r := vex-r **ffo**.

This completes the first step of structural instruction. The remaining steps are rather obvious:

$((ide-1 ; ide-2), ide-r=vex-r) =
    $(ide-1, ide-r=vex-r) ; $(ide-2, ide-r = vex-r)

$(**if** vex-b **then** sin-1 **else** sin-2 **fi**, ide-r = vex-r) =
    if vex-b **then** $(sin-1, ide-r = vex-r) **else** $(sin-1, ide-r = vex-r) **fi**

$(**while** vex-b **do** sin **od**, ide-r = vex-r) =
    while vex-b **do** $(sin, ide-r = vex-r) **od**

In short, after each assignment or a procedure call that changes the value of a register condition, we add a recovering assignment. The generalization of $ on specinstruction is rather evident.

In the end, let us point out a methodological difference between @ and $. The former is a character in the syntax of **Lingua-V**, and on the denotational side corresponds to a sequential composition of an instruction denotation with a data-expression denotation. Therefore:

Sde.[sin @ vex] = Sin.[sin] ● Sde.[vex]

In turn, $ is a constructor of syntaxes (from the level of **MetaSoft**)

$: Instruction x RegisterCondition ⟼ Instruction

where

RegisterCondition = Identifier = ValExp[86]

[86] Notice that the first sign of the equality belongs to MetaSoft and denotes the equality of formal languages, whereas the second — typed in Arial Narrow — is a character in the syntax of **Lingua**.

10 RELATIONAL DATABASES INTUITIVELY

10.1 Preliminary remarks

Section 11, which follows this one, is devoted to an extension of **Lingua** by selected database tools offered by SQL (Structured Query Language). Since we don't expect our reader to be familiar with SQL, the present section contains an informal description of some basic SQL-mechanisms that we shall try to formalize later. Several concept that we introduce in both mentioned sections do not appear in standard SQL manuals, and therefore they will be labelled by "(OWN)" which stands for "our OWN notion".

This section refers to several SQL sources since we didn't find a single manual sufficiently complete and unambiguous to identify the meaning of all these SQL mechanisms that we shall talk about. A nice book of Lech Banachowski [9] contains a model of relational databases and a description of SQL standard, but some issues are missing (e.g., three-valued predicates), and some others are only sketched. On the other end of the scale of clarity and preciseness is a thick volume of Paul DuBois [52]. We quote some "definitions" from that book just to show the scale of problems one has to tackle in building a denotational model for SQL**.** Between these two extremes, but certainly closer to DuBois, are four other books, [54], [60], [75] and [82]. Of course, since all the books mentioned above were published some time ago, certain mechanisms described there may look slightly differ today.

Lingua-SQL, whose draft denotational model will be given in Sec. 11, may be regarded as a sort of an API (Application Programming Interfaces) or a CLI (Call Level Interfaces)[87] on the ground of **Lingua**. API's have been created for such programming languages as C, PHP, Perl, Phyton, and CLI's — for ANSI, C, C#, VB.NET, Java, Pascal, and Fortran[88]. In each of these cases, a language is equipped with mechanisms allowing to run functionalities of an existing SQL engine. In our case the situation is different. If in the sequel anyone would undertake the challenge of implementing **Lingua-SQL** they should first implement their own SQL engine to make sure that this engine is adequate to our denotational model.

Similarly as in the case of **Lingua** we shall not attempt to defined a "completed" language. We shall only formalize some selected tools of SQL, to provide a denotational framework where a more complete SQL engine might be defined.

10.2 Basic values and their types

Types of only one category — table types — appear explicitly in SQL-manuals known to us. Several other types are present only implicitly. They include *basic types*[89] (OWN), i.e., the types of *basic values* (OWN) that appear in the fields of database tables, and *structural types* (OWN) such as the types of columns, rows and databases.

We shall define basic values as pairs consisting of a *basic data* (OWN) and a basic type. Basic data constitute probably one of the least standardised areas of SQL. Their categories may differ not only between different applications but also between different implementations of the same application.

In the present section, we base mainly[90] on [82], whose authors declare the compatibility with the standard ANSI SQL-2011[91]. The SQL syntax is printed, as in the former parts of the book, in Arial Narrow.

Database tables may be regarded as two-dimensional arrays carrying in their fields four sorts of basic values, each of them, except booleans, further split into several subcategories:

---

[87] CLI refers to the standard ANSI SQL (see [82] p. 359)

[88] Access has not been mentioned on these lists since it is available only together with Microsoft Basic Access.

[89] Some basic data are simple data, as defined in Sec. 4.1, but some others are not.

[90] „Mainly" but not „totally" since this manual also contains gaps.

[91] ANSI is an acronym of American National Standard Institute, and SQL-2011 is a standard accepted by ANSI in December 2011.

- **Numbers** split into two subsorts: *integers* and *decimal numbers* that split further into several types differing from each other by the range of values, e.g., SMALLINT, BIGINT or DECIMAL(p, s), where p (precision) denotes the maximal number of digits and s (scale) — the maximal number of digits after the decimal point.
- **Logical values** are handled as in the three-valued predicate calculus of Kleene (Sec. 2.10), and in [82] they are denoted by TRUE, FALSE, and NULL whereas in [54] by 0, 1, and NULL. Sometimes, e.g., in [60] instead of NULL we have UNKNOWN.
- **Strings** are, in principle, texts in our sense, but, similarly to numbers, they are split into subtypes depending on a maximal accepted number of characters. For instance, CHARACTER(n) is the type of words of the length n. The type of a string with varying length limited to n is called in [82] CHARACTER VARYING(n), and the type of a string of an unlimited length (whatever it means) is called BLOB. There exist also binary strings, and text-strings called TEXT.
- **Times** are tuples of three types: DATE — (year, month, day), TIME — (hour, minute, second), DAYTIME — (year, month, day, hour, minute, second).

Although it is nowhere explicitly said, one may guess (cf. [82]) that all sorts of data contain NULL that in some context plays the role of an abstract error. The majority of constructors, except boolean constructors, seem to be transparent for that error.

The constructors of basic data may be split into five following groups[92]:

1. Arithmetic operations: +, –, *, /.
2. String operations: CONCAT, UPPER, LOWER, SUBSTR, LENGTH.
3. Time operations: GETDATE, DAYNAME, DAYOFMONTH,
4. Basic predicates: =, <>, <, <=, >, >=, IS NULL, BETWEEN, LIKE.
5. Logical connectives: NOT, OR, AND.

The first group seems apparently quite obvious, but after a closer analysis we may find that it is obvious only in typical situations. E.g. 2+3 = 5, but if we try to add a number to a string (which is possible!), or to add two numbers whose sum exceeds the maximal allowed value, then the expected result is not clear. The source [82] does not comment on such cases at all, and in [52] p. 786, we can read the following[93]:

If we do not provide (…) correct values to functions, we should not expect reasonable results.

In another place of the same manual (p. 754) we read:

(…) *expressions that contain big numbers may exceed the maximal range of 64-bits computations in which case they return* <u>*unpredictable values*</u> (our emphasis).

We noticed that in the definitions of arithmetic operations, NULL does not appear, although it could be used as an abstract error. In this place, the worst possible solution has been chosen: instead of an error message, we have an "unpredictable result" which means that the computation does not abort, but generates a false result of an unpredictable value without warning the user.

Especially many unclarities are associated with default rules for type-conversion. For instance ([52] p. 753) the following rule concerns the addition operation if its arguments are words:

… *'+' is not an operator for the concatenation of texts, as it is the case in some programming languages. Instead, before the performance of the operation, textual strings are converted into numbers. Strings that* <u>*do not look like numbers*</u> (our emphasis) *are converted to 0.*

This rule is illustrated with the following examples:

'43bc' + '21d' = 64

---

[92] The descriptions of 1 to 4 are from [82] (pp. 129 and 180) and of 5 and 6 from [60] (pp. 191 and 201). The terminology is ours.

[93] Our own translation from the Polish version of the book.

'abc' + 'def' = 0

It hasn't been explained, if, e.g., '43ab2c' "looks like a number", and if it does, is it converted to 43 or 432? It has not been explained either, whether these rules apply to other arithmetic operations.

Fortunately [82] treats conversion a little more seriously — although still informally — introducing four types of conversions:

1. strings to numbers,
2. numbers to strings,
3. strings to dates and times,
4. dates, and times to strings.

String-operators offer fewer ambiguities, but still are defined only for typical situations. For instance, we did not find information about what happens if the concatenation of two strings exceeds an accepted length.

Time-operators offer further examples of inconsistencies between different SQL-applications that concern both the syntax and the types of operators. We shall not further analyse this problem since the involved operators are easy to formalise, once we decide about their meanings.

Predicates are typologically ambiguous since, in the majority of cases, they apply to all four sorts of data. E.g., the operators '=' and BETWEEN may be used for numbers and strings and probably also for dates. Their definitions are rather vague. E.g., in [82] p. 130, we can read:

If in a query, we use the (=) operator, the compared values must be identical, and in the opposite case, the condition is not satisfied.

It has not been explained if "not satisfied" means "false" or "not true". E.g. should we regard the value of the boolean expression 12 = abc as false or error?

The operator BETWEEN takes three arguments and checks if the first is between the second and the third in some default ordering.

The operator LIKE takes two string-arguments and checks if the first coincides with the pattern described by the second. Patterns are described using letters, digits and two special symbols:

% — an arbitrary string of characters (possibly empty)
_ — an arbitrary character

The only source where we found complete definitions of logical operators is [60], where a table-definition is given on page 191 and corresponds to Kleene's operators defined in Sec. 2.10. It seem rather strange that although we have a NOT operator in the language, special negated versions are introduced for all predicates, e.g., NOT NULL or NOT BETWEEN.

For all non-boolean operators, we have in SQL a situation which is typical for software manuals. Within the area of standard ranges of arguments, everything is clear. If, however, we go beyond this area, we can hardly predict what will happen. With a high degree of certainty, we may expect to encounter a different surprise in each implementation.

10.3 Creating tables

A central SQL-concept is a *table*, that is a two-dimensional array, but may be also regarded as a tuple of named columns where columns are tuples of basic values of a common type. Alternatively, a table may be regarded as a tuple of rows, where rows are mappings from identifiers (column names) to basic values. The intersections of rows and columns are called *table fields*. Of course, in reachable tables all rows have a common set of column names and tables are rectangular.

Tables in SQL are storable, i.e., assignable to variables in memory stores. In the sequel, variables carrying tables will be called *table variables* (OWN) or *table names*. To declare a table variable, we use operator

CREATE TABLE that assigns to a variable identifier a *table type* and (we can guess) a one-row table of default values indicated by table type[94].

A *table type* (OWN) may be seen as a tuple consisting of:

- a mapping assigning to each column name a column type,
- a predicate, called a *row yoke* (OWN), that describe a common property of all rows.

In turn, a *column type* (OWN) consist of:

- a basic type to be the common type of all values standing in the column,
- an (optional) default value or an indication that it can't be NULL,
- a predicate, called *column yoke* (OWN), that describes properties of the column,
- a finite set of *marks* (OWN) that describe relationships between tables in the database (Sec. 10.4).

Here is an example of two such declarations cited with only minor modifications after [9] p. 14[95]:

```
CREATE TABLE Affiliations
    (
    Department_ID       Number(3)                          PRIMARY KEY,
    Department          Varchar(20)  NOT NULL   UNIQUE
    City                Varchar(50)
    );


CREATE TABLE Employees
(
    Employee_ID         Number(6)                          PRIMARY KEY,
    Name                Varchar(20)  NOT NULL,
    Position            Varchar(9)   DEFAULT NULL,
    Manager             Number(6) ,
    Employment_date     Date,
    Salary              Number(8,2),
    Bonus               Number(8,2),
    Department_ID       Number(3)                          REFERENCES Affiliations,
CHECK(Bonus + Salary < 10000)
    )
```

The tabulation in this example shows a certain universal structure of a declaration:

- in the first column we see column names[96] of the future table; they will be common to all rows constituting this table,
- the remaining columns carry informations about data stored in table columns; in our model, they will be expressed by the mentioned already four components of a column type (some of them may be optional),

Special cases represent informations expressed by REFERENCES Affiliations and PRIMARY KEY that indicate a subordination relation between tables (Sec. 10.4),

---

[94] We did not find in the literature on SQL any information about what category of tables is assigned to a table-variable by its declaration.

[95] In Sec. 11 we shall frequently refer to this example and also to some other examples from [9]. In all cases we keep the original notation, where Number(p) denotes a type of total numbers with p digits, and Number(p, s) denotes the type of decimal numbers of the total number of digits equal to p and the number of digits after decimal point equal to s. In turn Varchar(n) denotes the type of strings of length not exceeding n.

[96] It may be slightly misleading at the beginning that in the syntax of table declaration column names are positioned vertically in one column rather than horizontally as they will appear in tables displayed on monitors. It is only a notational convention that facilitates writing table-type declaration with numerous column names.

In the last row of the second declaration we see a row-yoke expression describing the requirement that the values of the fields Salary and Bonus in each row of the future table satisfy the indicated condition.

The elements of a table declaration, except column names, will be referred to as *integrity constraints of a table*. Their meanings are following:

1. Number(3) — the type of data in the column.
2. DEFAULT — a default value follows this keyword.
3. NOT NULL— all fields in the column must not be empty, i.e., none of them may be NULL. An attempt of a violation of this constraint should generate an error signal.
4. UNIQUE — no two identical data may appear in the column. If this happens, an error message should be raised.
5. PRIMARY KEY — carries two pieces of information: (a) that this column may be a parent column for a column of another table (see Sec. 10.4), (b) that repetitions of elements are not allowed.
6. REFERENCES Affiliations — the field Department_ID in table Employees is related to the field of the same name in the table Affiliations. Relations between tables are used to modify tables and to create queries.
7. CHECK(Bonus + Salary < 10000) — all rows of the table should satisfy this condition.

As we see from this example, when we declare a table variable, we define its type[97]. In this way we define seven groups of properties of a future table:

1. the names of columns, e.g., Department_ID,
2. the types of data in all fields of a given column, e.g., Number(6),
3. the default value for a given column, e.g., DEFAULT NULL,
4. restrictions concerning columns as a whole, e.g., NOT NULL or UNIQUE,
5. indicator of a special role of a column in the database, e.g., PRIMARY KEY,
6. relationships between tables by indicating related columns in tables, e.g., REFERENCES Affiliations.
7. relationships between values in each row, e.g., CHECK(Bonus + Salary < 10000); in our model it will be called a *row yoke* (Sec. 11.2.3).

10.4 Databases and subordination relation between tables

By a *database* we shall mean a finite collection of named tables, i.e. a mapping from identifiers to tables. A *subordination relation* in a database may be defined as a set of triples of the form

(table name 1, column name, table name 2)

including two (different) names of tables in the base and a common name of their columns. Denotationally these relations may be regarded as yoks that define properties of databases.

The mechanism of establishing relations between tables appears in SQL literature in several versions. All of them base on a common idea, although their implementations may be different. Below we try to describe this common idea.

Consider tables Affiliations and Employees from Sec. 10.3. In Employees, we have a column Department_Id which defines the association of an employee to a department. In its declaration we have marking REFERENCES Affiliations expressing the fact that in the table Affiliations we may find information about the department where an employee is employed. Instead of storing in the table Employees the information about the departments where they works, we only show the ID's of these departments that identify appropriate rows in the table Affiliations. Now, for this construction to have a practical sense, our two tables must satisfy three conditions:

[97] It seems that SQL lacks a mechanism that would allow to define a table type as a stand-alone element, i.e., independently of a variable declaration.

1. the column Department_ID must appear in both tables,
2. every ID of a department which is in the table Employees must also appear in the table Affiliations (but not necessarily vice versa),
3. in Affiliations the column name Department_ID must have no repetitions.

If these conditions are satisfied, then we say that:

the column name Department_ID *links the tables* Affiliations *and* Employees *with a subordination relation.*

In the pair of tables, Affiliations, and Employees, the table Affiliations is called a *parent* table or a *superior table,* whereas Employees is a *child* table or a *subordinated* table. The column name Department_ID is a *primary key* in Affiliations and a *foreign key* in Employees.

If an employee's row ER and a department's row DR have the same value in the field Department_ID, then we say that the ER *points* to the DR (OWN).

The establishment of a subordination relation between tables has consequences for operations on these tables. For instance:

- Introducing an employee who has been employed in a non-existent department should be impossible, i.e. the database-engine should generate an error message in that case.
- A department's record cannot be removed from a table until there are employees employed in that department. An alternative solution is that all employees of the removed department are automatically removed by the engine; a *cascading* solution.
- One can request the creation of a table with three columns that combine information from both linked tables, e.g., with columns Name, Department, City.

10.5 Instructions of table modification

Tables that have been declared may be modified by instructions. Below we show a few typical examples:

Entering a new column to a table:

```
ALTER TABLE Employees
ADD COLUMN ID_number CHAR(11) DEFAULT NULL
```

We add a column to a table, and we indicate a default value for that column.

Deleting a column from a table

```
ALTER TABLE Affiliations
DROP COLUMN Department_ID CASCADE
```

This instruction is executed with the option CASCADE, which means that the deletion of a column results in the deletion of all columns in the tables of current database that refer to that column. An alternative option is RESTRICT, where the instruction is not executed whenever such columns exist in the database.

Notice that the instructions from the group ALTER TABLE modify not only the content (the data) of a table but also its type. There are other examples of instructions altering tables ([60] p. 49):

- ALTER COLUMN — column-type is modified by SET DEFAULT or DROP DEFAULT, which sets or drops a default value.
- ADD — new constraint is added to an existing column.
- DROP CONSTRAINT — the removal of a constraint from an indicated column. For this instruction, RESTRICT or CASCADE must be set.

Another group of table-modifying instructions change the contents of tables without modifying their type. Some typical examples are:

The insertion of a new record (row):

```
INSERT INTO Affiliations
VALUES (095, ‘Marketing’, ‘London’)
```

This instruction may also be written in a form where column names are explicit (cf. [54], p. 73)

```
INSERT INTO Affiliations (Department_ID, Dep_name, City)
VALUES (095, ‘Marketing’, ‘London’)
```

In both examples the row-oriented conditions (the row yokes) have been dropped, which means that they are tautologies. However, if this is not the case, these conditions are not modified by this instruction, i.e., the new row has to satisfy them.

A conditional modification of data in one column. E.g., the increase of salaries of all salesmen by 10%:

```
UPDATE Employees
SET Salary = Salary * 1,1
WHERE Position = ‘salesman’
```

The removal of all rows that satisfy a given yoke. E.g., the removal of all employees who have no position:

```
DELETE FROM Employees
WHERE Position IS NULL
```

A particular situation takes place if we drop a row with a primary key which is a foreign key in a child-table, e.g.:

```
DELETE FROM Affiliations
WHERE Dep_name = ‘production’
```

If in the child table Employees the key Department_ID is — as in our case — a foreign key and there exist rows which point to the rows that are supposed to be deleted from Affiliations, then the operation is not executed and an error message is generated. However, the operation:

```
DELETE FROM Affiliations
WHERE Dep_name = ‘production’ CASCADE
```

will be executed, and additionally, in the table Employees, all rows that point to the row, which is deleted from Affiliations, are deleted as well[98].

10.6 Transactions

By a *transaction,* we mean a sequence of instructions closed (or not) in some parentheses such as, e.g., BEGIN TRANSACTION and COMMIT TRANSACTION[99]. A transaction may be equipped with an error recovery mechanism that stops the execution of a transaction whenever:

- the execution would violate integrity constraints, or
- the execution is not possible, e.g., we search for a non-existing element in a table.

In all such cases, the implementation returns to the initial database state of the transaction that is called the *roll-back value of the database*[100].

Five following keywords are used to control the recovery mechanism of transactions in SQL-programs:

SAVEPOINT — save rollback-value of a database
RELEASE SAVEPOINT — delete rollback-value

[98] There is a certain inconsistency in SQL compared with the deletion columns. In the case of rows option RESTRICT is set by the system without the possibility of choosing another option by the user.

[99] These parentheses may differ between applications (some manuals are not mentioning them at all). Here we use the notation of Bena Forty ([54], p. 175) which is a standard for Microsoft SQL Server.

[100] We have to warn the reader that in all known to us manuals, transactions are described in an exceptionally unclear and incomplete way, and therefore our understanding of this construction is based more on guesses than on facts.

ROLLBACK — call-of transaction
IF — a conditional activation of a rollback
COMMIT TRANSACTION — accept transaction.

The instruction

SAVEPOINT savepoint-name

assigns the actual database to a temporary user-defined database name (variable) savepoint-name. The instruction

RELEASE SAVEPOINT savepoint-name

deletes the variable savepoint-name (and its value) from the state. The instruction

ROLLBACK savepoint-name

brings the database to its rollback-value and deletes the variable savepoint-name. This instruction may also appear without a parameter, in which case the database is (probably?) rolled back to the value initial of transaction-execution[101]. In such cases, the execution of a transaction should start with a default SAVEPOINT, which saves database value to some system variable. It also seems that ROLLBACK aborts program execution and generates an error message.

To make the execution of ROLLBACK dependent on an error message, one may use the conditional IF constructor. Ben Forta ([54] p. 179) shows the following example:

IF @@ERROR <> 0 ROLLBACK savepoint-name

It is explained there that @@ERROR is a system-variable whose value equals 0 if there is no error message, and (we guess) equals some error message (or 1?) in the opposite case.

This example suggests — although that hasn't been explicitly written — that the condition of IF might be of the form

@@ERROR = error-message

with a specific error message. Such a solution would allow making the execution of ROLLBACK dependent on the type of an error.

The execution of COMMIT results in saving the result of the transaction and deleting all earlier declared rollback-variables.

For instance, in a database carrying data of bank customers, the transaction that moves 1000 $ from one account to another may have the following form:

```
BEGIN TRANSACTION

   SAVEPOINT start

   UPDATE Accounts
   SET Balance = Balance – 1000
      WHERE ClientID = 1250 ;

   IF @@ERROR <> 0 ROLLBACK start ;

   UPDATE Accounts
   SET Balance = Balance+ 1000
      WHERE ClientID = 1260 ;

   IF @@ERROR <> 0 ROLLBACK start
```

[101] The parameter-less version of this instruction appears in the majority of manuals known to us.

```
COMMIT TRANSACTION
```

The first ROLLBACK takes place if there is no customer in the database with ID equal 1250, or if its balance-value is less than 1000. The second ROLLBACK is activated if the first is not, but there is no customer in the database with ID equal 1260.

Notice that after the execution of the first UPDATE, the actual sum of all deposits is not equal to the bank-balance of deposits, which means that the integrity constraints are violated. The second UPDATE "removes" this violation, but if it can't be performed because of the lack of 1260-customer, then the transaction would end with an inconsistent database. The second ROLLBACK prevents such a situation.

10.7 Queries

Queries are used to collect information from one or more tables in the form of a new table. The execution of a query results in the generation of a table and possibly in displaying it on a monitor. Queries are constructed by several variants of operator SELECT. Below a few typical examples:

The selection of indicated columns of a table:

```
SELECT Name, Salary, Position
FROM Employees
```

As a result of this query, a monitor displays a three-column table with columns indicated by the parameters of SELECT.

The selection of columns combined with the filtering of rows:

```
SELECT Name, Salary, Position
FROM Employees
WHERE Department_ID = 10
```

In WHERE clause, we may have boolean expressions with operators on basic data described in Sec. 10.2.

Queries may be composed of other queries using operators called by Banachowski [9] *algebraic operators on queries*. These operators may be applied to more than one table. For instance:

```
SELECT Department_ID
FROM Affiliations
EXCEPT
SELECT Department_ID
FROM Employees
```

This query generates a one-column table of the ID's of these departments that appear in the table Affiliations but that do not appear in the table Employees. i.e., the ID's of departments with no employees.

A specific group of queries allows reaching more than one table. In such a case, we say that queries use the *joins of tables*. Below we see an example of a query that selects data from two tables — Employees and Affiliations.

```
SELECT Employee_ID, Name, Department_ID
FROM Employees, Affiliations
WHERE Employees.Department_ID = Affiliations.Department_ID
AND Affiliations.City = 'London'
```

This query generates a three-column table where each row contains the ID of an employee, his/her name, and the name of the department where he/she is employed. The condition in WHERE-clause is called a *joint predicate*. In our case, it returns only such rows where employees are employed in departments located in London.

In WHERE-clauses, we may use boolean expressions exploring basic predicates on basic data (Sec. 10.2), e.g.:

```
SELECT Employee_ID, Name, Salary
FROM Employees
WHERE Salary > 1000  AND  Salary <= 2000
```

or set-theoretic operators. For instance, the query:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Position IN ('cashier', 'salesman', 'manager').
```

generates a table with cashiers, salesmen, and managers. The query:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Salary > ALL
(
SELECT Salary
FROM Employees
WHERE Position = 'cashier'
)
```

generates a table that shows employees with salaries higher than the salaries of all cashiers. In this case, we have to do with a *nested query,* where the inner SELECT generates a column with the salaries of all cashiers. Let us denote:

sae : SalEmp — the set of values in the column Salary of the table Employees,
sac : SalCas — the subset of SalEmp that contains the salaries of cashiers,
shc : SalHigCas — the subset of SalEmp that contains salaries higher than the salaries of cashiers

In this case:

SalHigCas = { sae | sae : SalEmp **and** (∀ sac : SalCas) sae > sac }

where > is a predicate that compares numeric values and generates an error whenever at least one of its arguments is not a numeric value.

The transparency of > implies that the set SalHigCas contains numbers only, although it may be empty as well. In particular, it is empty if SalCas contains at least one not-number.

In no bibliographic sources we found information what happens if inequality sae > sac generates an error. Will it interrupt a program and generate an error, or the query will generate some "unexpected" table, maybe empty?

Let us consider now a query that results from the former, if ALL is replaced by EXISTS, i.e., that generates the table of employees with salaries higher than the salary of at least one cashier[102]:

```
SELECT Employee_ID, Name, Position, Salary
FROM Employees
WHERE Salary > EXISTS
(
SELECT Salary
FROM Employees
WHERE Position = 'cashier'
)
```

Denote:

shs : SalHigSomCas — salaries higher than some salaries of cashiers.

[102] In this case we use a syntax which is — maybe — not compatible with SQL. we used it, however, to keep the similarity with the ALL example, whose syntax (although not the example itself) has been taken from p. 139.

In that case:

SalHigSomCas = { sea | sea : SalEmp and (∃ sac : SalCas) sea > sac }

hence:

SalHigSomCas = { sea | sea : SalEmp and exists.(SalCas, >).sac = tt }

In that case, contrary to the former, if SalCas contains not-numbers, then the set SalHigSomCas does not need to be empty.

Notice now that whenever the evaluation of sae > sac for some sac, generates an error, then

exists.(SalCas, >).sac = ff

If, however, we replace EXISTS by SOME, then an error may appear. This replacement does not change the table generated by our query but affects error generation.

Quantifiers may also appear in the context of joining tables. The query shown below generates the table of departments where at least one employee is employed.

```
SELECT Department_ID
FROM Affiliations
WHERE Department_ID = EXISTS
(
SELECT Department_ID
FROM Employees
)
```

As was mentioned in Sec. 10.2, for every simple operator, there exists its negated version, e.g., = and <>, LIKE and NOT LIKE, etc. Similarly, we have NOT IN. In the case of set-theoretic quantifiers, we have found only NOT EXISTS and only in [82] p. 147 and in [52] p. 242. Of course, none of these sources concerns the case where EXISTS generates an error.

10.8 Views

If we want to use a query more than once, we may declare it as a procedure. Such procedures are called *views*. Below we see an example of a view-declaration:

```
CREATE VIEW Officials
   (Employee_ID, Name, Salary)
   AS SELECT Employee_ID, Name, Salary
   FROM Employees
   WHERE Position = 'official'
```

This view is named Officials and creates a three-column table by selecting columns from Employees and rows with 'official' that stands in the column Position.

Since views are procedures, they have no counterparts in syntax (cf. Sec. 6.6.1). At the syntactic level, we only have *view declarations* CREATE VIEW and *view calls* (OWN) that refer to the names of views.

View calls may be used in queries in the same way as tables and, of course, a view is executed in the call-time state rather than in the declaration time state. In SQL-manuals, views are, therefore, referred to also as *virtual tables*. Views may also be called in instructions that create or modify tables. Consider the following view-declaration:

```
CREATE VIEW Salesmen
   AS SELECT * FROM Employees
   WHERE Department_ID = 20
```

In this declaration, the star "*" means that we chose all columns, and the number 20 is the ID of the sales department. Calling the view Salesmen we can create an instruction that modifies the table Employees by increasing the salaries of all salesmen by 10%:

```
UPDATE Salesmen
    SET Salary = Salary * 1,1.
```

In the case of using vies for the modifications of tables, each SQL engine has its specific restrictions. E.g., MySQL requires that in SELECT-clauses, only column names may appear.

A special case are *views with check option* which force the checking of a condition when views are used in instructions. Banachowski [9] shows an example of such a view:

```
CREATE VIEW Employees_on_not-payed_holiday
    AS SELECT *
    FROM Employees
    WHERE Salary = 0 OR Salary IS NULL
    WITH CHECK OPTION
```

If this view is used in the instruction:

```
UPDATE Employees_on_not-payed_holiday
    SET Salary = 1000
    WHERE Name = 'Smith'
```

then it is not executed if the salary of Smith is 0 or NULL.

10.9 Cursors

Cursors are used to assign selected rows of tables to value variables. This mechanism allows for processing databases using programs written in user-interface programming languages such as API or CLI (see Sec. 10.1). A cursor points to a row in an indicated table and allows us to get data from that row. Tables indicated by cursors are defined using queries. As a matter of fact, we should not talk about a cursor as such, but about a *cursor of a table*, or maybe about a *cursor of a query.*

Cursors are created using *cursor declarations,* which assign a cursor to a *cursor name* (an identifier). Such declarations are of the form[103]:

```
DECLARE cursor_name IS
    SELECT ...
```

After a cursor has been declared, it is not yet ready for use. To make it ready, we have to apply an *opening instruction* of the form:

```
OPEN cursor_name.
```

This instruction causes the execution of SELECT, which appears in the declaration and (we guess) in the setting of the so-called *cursor grasp* at the "position" preceding the first row of the generated table. The operation of getting data from a table is:

```
FETCH NEXT cursor_name INTO variable
```

The NEXT means getting the data of the row next to the grasp and moving the grasp one row further. It seems, therefore, that OPEN sets the cursor before the first row.

The FETCH NEXT instruction is usually applied in a program loop, which means that when a grasp reaches the last row of a table, it can't be moved further. We have found only one comment on that issue in [82] p. 353 (our translation from the text in Polish):

---

[103] The syntax of a cursor-declaration depends upon application. Here we use the syntax of ORACLE ([82] p. 352).

In every implementation of databases, cursors are implemented in a slightly different way, but each of them enables a correct cursor-closing without an unnecessary generation of errors.

If a cursor is temporarily not needed, we close it by instruction:

```
CLOSE cursor_name
```

This instruction leaves the cursor structure for reopening.

10.10 The client-server environment

So far, when talking about SQL-systems, we were assuming tacitly that the user has a database to his/her exclusive disposal. However, that is usually not the case. In general, there may be more than one user, which means that we need tools to give them or to deny access to databases. Here is an instruction scheme which sets a lock on a given table:

```
LOCK TABLE table_name
   IN [SHARE | EXCLUSIVE]
   [NOWAIT]
```

where the options in square-brackets mean the following:

- SHARE — the lock applies to all users,
- EXCLUSIVE — the lock applies to all users except the one who sets the lock,
- NOWAIT — do not wait for lock setting, if it can't be set at the moment.

Locks are removed by instructions COMMIT or ROLLBACK. An example of an instruction which gives permissions to a given user may be:

```
GRANT SELECT, UPDATE (Salary)
   ON Employees
   TO Smith
```

This instruction grants the permission of performing SELEC and UPDATE in the table Employees to the user Smith.

These mechanisms of SQL may differ between the application, but since they are relatively simple to describe, we shall not discuss them later.

Może należałoby usunąć dwa ostatnie rozdziały, skoro nie będziemy formalizować opisanych w nich mechanizmów. ???

11 A DENOTATIONAL MODEL FOR DATABESES: Lingua-SQL

11.1 Lingua-SQL as an enrichment of Lingua

We shall build **Lingua-SQL** as an enrichment of **Lingua** by new categories of types, values, yokes, denotations and corresponding constructors. However, the algebra of denotations of the new language will not be an algebraic extension of the former algebra, i.e., will not include former caries and constructors plus some new ones (see Sec. 2.12). That is why we use here the term "enrichment" rather than "extension". Formally, the new algebra will include inherently new carriers and constructors. Some of them will be, in a sense, replicas of the formers, and some others will correspond to "genuinely new" mechanisms coming from an SQL engine.

The borderline between **Lingua** and **Lingua-SQL** will be explicitly seen in the definition of *new states* that will be pairs consisting of:

- a state in the former sense, called a *hereditary component,*
- a new *SQL component* carrying SQL structured values, i.e., *tables* and *databases*.

Databases will carry tables, and tables will carry basic values. We shall assume that the SQL basic values will constitute subcategories of typed data of **Lingua** and therefore will be generated by (formerly defined) value expressions.

We shall not introduce expressions evaluating to tables or databases and consequently we will not have assignment instructions assigning structured SQL values to variables. Tables and databases will be created by declarations, and further developed, enriched or modified by instructions. They will be assigned to their variables directly, rather than via references.

In the category of specific SQL types we will have three subcategories:

- basic types,
- column types,
- table types.

Column and basic types will include yokes which is a technical consequence of the fact that we will not have references in the SQL components of states. We do not introduce database types since they will be implicit in the types of their tables.

In building **Lingua-SQL** we will ensure the satisfaction of two *adequacy principles* relating our model to "typical implementations" of SQL:

1. whenever a typical implementation raises an error message, our model should guarantee the same,
2. whenever in a typical implementation, "one can't expect a meaningful result" (cf. Sec. 10.2), our corresponding constructor should raise an appropriate error message.

Similarly, as in the case of **Lingua,** we do not pretend to define a "practical" language. Our goal is to identify selected critical challenges in building a denotational framework for database mechanisms, but certainly not to fully cover the SQL diversity of tools. We may also see our experiment with SQL as a case study of adding inherently new mechanisms to an existing programming language.

For compactness, we shall refrain from discussing the syntax of **Lingua-SQL**. We believe this task should be reasonably evident for the readers who went through Sec. 7.

11.2 Data, types, values and states

11.2.1 Basic data, types and values

The domains of basic data are the following:

bad : BasDat = Boolean | SmaInt | BigInt | Decimal.(p, s) | String | Date | Time | DateTime | {Ω} basic data
bad : Boolean = {true, false} boolean data

bad : SmaInt = … small integers
bad : BigInt = … big integers
bad : Decimal.(p, s) = Integer.(p − s) x Integer.s decimal numbers with s < p
bad : Integer.n = … integers with decimal representations of length n

bad : ChaData = Character.n | CharVar.n | Blob character data
bad : Character.n = Sign^{cn} words of length n
bad : Sign = … a set of signs
bad : CharVar.n = Character.1 | … | Character.n words of length not larger than n
bad : Blob = CharVar.m where m is a parameter of the model

bad : Date = Year x Month x Day
bad : Time = Hour x Minute x Second
bad : DateTime = Date x Time

bad : Year = {0,…,9999)
bad : Month = {1,…,12}
bad : Day = {1,…,31}
bad : Hour = {0,…,23}
bad : Minute = {0,…,59}
bad : Second = {0,…,59}

The element Ω represents an empty field of a table and is called an *empty data* (OWN).

For simplicity we assume that all basic data except time-and-date data are included in the corresponding domains defined in Sec. 4.1, i.e. in Integer, Real and Text. We also assume that all constructors of simple data are applicable to basic data. In the time-and-date data category, we assume to have some typical constructors, but we shall not define them explicitly. We may regard them as parameters of our model.

Basic types, i.e., the types of basic data are defined by the following equations:

bat : BasTyp = BooTyp | IntTyp | DecTyp | ChaTyp | TemTyp *basic types*

bat : BooTyp = {'boolean'} *boolean type*

bat : IntTyp = SmaIntTyp | BigIntTyp *integer types*
bat : SmaIntTyp = {'smaint'} *small-integer types*
bat : BigIntTyp = {'bigint'} *big-integer type*
bat : DecTyp = {'De'} x SmaInt x SmaInt *decimal type*

bat : ChaTyp = {'Ch'} x SmaInt | {'ChV'} x SmaInt | {'blob'} *character types*

bat : TemTyp = {'date', 'time', 'datetime'} *date-and-time type*

We skip obvious definitions of basic type constructors. The clanning function

CLAN-bt : BasTyp ⟼ Sub.BasDat

is defined analogously as in **Lingua**. We assume additionally that Ω is of any type, i.e., Ω : CLAN-bt.bat for any bat : BasTyp. By a *basic value* we mean a basic data and its type:

bav : BasVal = {(bad, bat) | bad : CLAN-bt.bat}

Basic values of the form (Ω, bat) are called *empty values*. The constructors of basic types are defined analogously to the constructors of simple types (Sec. 4.2). We skip their definitions. For technical convenience we introduce two projection functions:

type : BasVal ⟼ BasTyp
data : BasVal ⟼ BasDat

Their definitions are obvious.

11.2.2 Columns, their yokes and types

By a *column* we mean a nonempty tuple of basic values of a common type:

col : Column = {(bav-1,…,bav-n) | n ≥ 1 **and** (∀i,j)(type.bav-i = type.bav-j)}

For technical convenience we extend the formerly defined function type to columns:

type : Column ⟼ BasTyp

We do not introduce constructors of columns since we will not need them. By a *column yoke* we mean a predicate on columns:

coy : ColYok = Column ⟼ BooValE

Note that in contrast to yokes in **Lingua** defined in Sec. 4.4, column yoke return only boolean values or errors (an engineering decision).

Column yokes may be *simple* or *composed.* Composed yokes are propositional compositions of simple yokes. Similarly as in Sec. 4.4, propositional connectives in column yokes are Kleene's connectives (Sec. 2.10). Simple column yokes are again split into two subcategories:

- *quantified column-yokes* — describing common properties of all elements of a column; e.g., that all elements are greater than 10,
- *holistic column-yokes* — describing properties of columns "as a whole"; e.g., that a column is ordered increasingly or that it is free of repetitions.

The names of constructors of column yokes will be prefixed by qcy for quantified yokes and by hcy for holistic yokes. Similarly to yokes in **Lingua**, also column yokes will become the denotations of corresponding expressions. Typical examples of column yoke constructors may be the following (in- stands for "integer"):

qcy-greater-in	: BasVal	⟼ ColYok	
qcy-less-in	: BasVal	⟼ ColYok	
qcy-equal	: BasVal	⟼ ColYok	
qcy-later	: BasVal	⟼ ColYok	
qcy-nonempty	:	⟼ ColYok	
…			
hcy-ordered-lex	:	⟼ ColYok	lexicographically ordered
hcy-unique	:	⟼ ColYok	no repetitions
…			
coy-and	: ColYok x ColYok	⟼ ColYok	conjunction of yokes

Let's see the definition of the first quantified constructor. It builds a yoke which is satisfied if every data in a column is of an integer type and is greater than a given integer:

qcy-greater-in : BasVal ⟼ ColYok i.e.
qcy-greater-in : BasVal ⟼ Column ⟼ BooValE
qcy-greater-in.bav.col =
let
((bad-1, bat),…,(bad-n, bat)) = col
(bad, v-bat) = bav
bat /: IntTyp ➔ 'integer type expected'
(∀1≤i≤n)(greater-in.(dat-i, bad) ➔ tv
true ➔ fv

Here greater-in denotes a comparison relation of integers from an implementation platform (see Sec. 4.1). The definition of the first holistic constructor is the following:

hcy-unique : ⟼ ColYok
hcy-unique : ⟼ Column ⟼ BooValE
hcy-unique.().col =
no-repetitions.col ➔ tv
true ➔ fv

To define column types we first introduce a domain of *column markings*

com : ColMar = FinSub.(Identifier | {'primary'}) column markings

A column marking is a finite, possibly empty, set of identifiers plus possibly the mark 'primary'. We need three constructors to build them:

com-build-empty : ⟼ ColMar
com-add-primary : ColMar ⟼ ColMar
com-add-ide : Identifier x ColMar ⟼ ColMar

We skip their obvious definitions. By a *column type* (OWN) we mean a quadruple

(bat, bav, yok, com) : BasTyp x BasVal x ColYok x ColMar

such that

type.bav = bat

The basic value bav is called the *default value* of the column type. By

cot : ColTyp column types

we denote the domain of column types. Note that, again in contrast to **Lingua** types, column types include yokes. Technically it is the consequence of the fact that we do not use references where yokes in **Lingua** are located. Column types will be built by only one constructor:

ct-create : BasTyp x BasVal x ColYok x ColMar ⟼ ColTypE
ct-create.(bat, bav, coy, com) =
type.bav ≠ bat ➔ 'wrong type of default value'
true ➔ (bat, bav, coy, com).

We skip simple definition of column marking constructors. The *clans of column types* are defined as sets of columns:

CLAN-ct : ColTyp ⟼ Set.Column

such that (bav-1,…,bav-n) : CLAN-ct.(bat, bav, coy, com) iff:

(1) type.(bav-1,…,bav-n) = bat — the common type of all values of the column is the type indicated by the column type

(2) if data.bav ≠ Ω, then (∀i)(data.bav-i ≠ Ω) — if the default value of the column is not empty, then all values of this column must not be empty, although they do not need to be the default values,

(3) coy.(bav-1,…,bav-n) = tv — the column satisfies the column yoke,

(4) 'primary' /: com **or** no-repetitions.(bav-1,…,bav-n) — there are no repetitions in a parent column

Practical implications of (4) will be explained in Sec. 11.2.5.

In the definitions of future table constructors we shall use the following function to check if a given column is of a given type. Note that this function not only checks the type-correctness of a column, but also identifies categories of type violations whenever they occur. This fact is an important feature of our error detection mechanism.

```
check-column-type : Column x ColTyp ⟼ {'OK'} | Error
check-column-type.(col, cot) =
   let
      (bav-1,…,bav-n)       = col
      (bat, bav, coy, com)  = cot
   type.col ≠ bat                              ➔ 'type inconsistency'        for i = 1;n
   data.bav ≠ Ω  and data.bav-i = Ω            ➔ 'value must not be empty'   for i = 1;n
   coy.col : Error                             ➔ coy.col
   coy.col = fv                                ➔ 'column yoke not satisfied'
   'primary' : com and are-repetitions.col     ➔ 'repetitions in a parent column'
   true                                        ➔ 'OK'
```

Of course

CLAN-ct.cot = {cod | check-column-type.(cod, cot) = 'OK'}

In the sequel we shall use the following auxiliary projection function:

```
marking : ColTyp ⟼ ColMar
marking.(bat, bav, coy, com) = com
```

11.2.3 Labeled rows and row yokes

Since row yokes will refer to column names, as e.g., in CHECK(bonus + salary < 10000), to define them we introduce a concept of *labeled rows* (OWN):

lar: LabRow = Identifier ⟹ BasVal | {Ω} labeled row[104]

Labeled rows will be also used in the definitions of table constructors. *Row yokes* are functions that given a labeled row return a basic value or an error:

roy : RowYok = LabRow ⟼ BasValE row yokes

Row yokes are used to describe common properties of all rows of a table. To make the reachable part of the domain of row yokes rich enough, we allow them to return arbitrary basic values, rather than just boolean values (cf. Sec. 4.4).

[104] We shall not introduce anything like "row values" since we shall not need them; besides, labeled rows carry values rather than data.

Row yokes will become the denotations of row-yoke expressions, and the latter will be similar to simple-value expressions. In their case, however, column names will stand in the place of variables, and they will point to basic values directly rather than via references[105]. Constructors of row yokes refer to (call) simple-value constructors and possibly some other special constructors otherwise not available at the level of expressions.

Constructors of row yokes are similar to constructors of value-expression denotations. E.g., constructors refereeing to addition and comparison of integers have the following signatures:

ry-add-int : RowYok x RowYok ⟼ RowYok
ry-less-int : RowYok x RowYok ⟼ RowYok

The definition of the first constructor is the following:

```
ry-add-int  : RowYok x RowYok ⟼ RowYok                         i.e.
ry-add-int  : RowYok x RowYok ⟼ LabRow ⟼ BasValE
ry-add-int.(roy-1, roy-2).lar =
   roy-i.lar : Error      ➔ roy-i.lar             for i = 1,2
   let
      bav-i           = roy-i.lar                 for i = 1,2
      (dat-i, typ-i)  = bav-i                     for i = 1,2
   typ-i /: IntTyp        ➔ 'integer expected'    for i = 1,2
   true                   ➔ td-add-int.(bav-1, bav-2)
```

In this definition td-add-int is an integer addition of typed data (Sec. 4.3). Note that it may generate an error message.

11.2.4 Tables and their types

Tables are central concept in SQL. Intuitively our tables will consist of a table type and a two-dimensional array of basic values called a *table-content*. However, depending on a table constructor, which we shall apply to a table, we shall regard table content as a tuple of labeled columns called a *column table-content* (OWN) or a tuple of labeled rows called a *row table-content* (OWN). To formalize these two perspectives of seeing a table we introduce two following domains:

ctc : ColTabCon = Identifier ⇒ Column column table-contents
rtc : RowTabCon = $LabRow^{c+}$ row table-contents

By a *rectangular column table-content* we mean a column table-content where all columns are of the same length. In an analogous way we define *rectangular row table-content* and we introduce two corresponding domains:

ctc : ReColTabCon rectangular column table-contents
rtc : ReRowTabCon rectangular row table-contents

By the *depth of a column table-content,* in symbols

depth.ctc

we mean the common length of its columns.

By a *table header* (OWN) we mean a nonempty mapping assigning column types to column names. By a *table type* (OWN) we mean a pair consisting of a table header and a row yoke.

---

105 We do not introduce reference in the SQL-part of our model since database value will not appear in objects. We return to this issue in Sec. 11.2.6.

tah : TabHea = Identifier ⟹ ColTyp table headers
tat : TabTyp = TabHea x RowYok table types

Table types are built by three constructors. The first of them creates a one-column table header.

create-tab-hea : Identifier x ColTyp ⟼ TabHea
create-tab-hea.(ide, cot) = [ide/cot]

The second constructor adds a new column to an existing table header:

add-to-tab-hea : TabHea x Identifier x ColTyp ⟼ TabHea
add-to-tab-hea.(tah, ide, cot) =
tah.ide = ! ➔ 'column name already exists'
true ➔ tah[ide/cot]

The third constructor creates a table type by adding a row yoke to a table header:

create-tab-typ : TabHea x RowYok ⟼ TabTyp
create-tab-typ.(tah, roy) = (tah, roy).

Given a table type with n columns:

tat = ([ide-1/cot-1,…,ide-m/cot-n], roy) where
cot-i = (bat-i, bav-i, coy-i, com-i) for i = 1;n

by the *clan* of this type, denoted by CLAN-tt.tat, we mean the set of all rectangular column table-contents with n columns named by the ide-i's of tat, i.e.,

ctc = [ide-1/col-1,…,ide-n/col-n] where
col-i = (bav-i1,…,bav-ik) for i = 1;n and some common k ≥ 1

such that

(1) each column is of the corresponding column type , i.e.
col-i : CLAN-ct.cot-i for i = 1;n,

(2) all labelled rows of the table satisfy the row yoke of the table type, i.e.
roy.[ide-1/bav-1j,…,ide-n/bav-nj] = tv for j = 1;k

By a *table* we shall mean a pair consisting of a rectangular column table-content and its type:

tab : Table = {(ctc, tat) | ctc : CLAN-tt.tat} *tables*

Similarly as in the case of column types, also now we define a function that checks if a column table-content is of a given table type:

type-table-check : ColTabCon x TabTyp ⟼ {'OK'} | Error
type-table-check.(ctc, tat) =
let
[ide-1/col-1,…,ide-n/col-n] = ctc
(bav-i1,…,bav-ik) = col-i for i = 1;n
[ide-1/cot-1,…,ide-n/cot-n] = tat
[ide-1/bav-1j,…,ide-n/bav-nj] = lar-j for j = 1;k
check-column-type.(col-i, cot-i) ≠ 'OK' ➔ check-column-type.(col-i, cot-i)
roy.lar-j : Error ➔ roy.lar-j for j = 1;k
roy.lar-j = fv ➔ 'row yoke violated'
true ➔ 'OK'

In the sequel we shall use two following functions to "switch" between two perspectives of seeing tables:

C2R : ReColTabCon ⟼ ReRowTabCon column-to-row conversion
C2R.[ide-1/col-1,…,ide-n/col-n] =
let
(bav-1i,…,bav-ki) = col-i for i = 1;n
lar-j = [ide-1/bav-j1,…,ide-n/bav-jn] for j = 1;k
true ➔ (lar-1,…, lar-k)

R2C : ReRowTabCon ⟼ ReColTabCon row-to-column conversion
R2C.(rod-1,…,rod-k) =
let
[ide-1/bav-j1,…,ide-n/bav-jn] = lar-j for j = 1;k
col-i = (bav-1i,…,bav-ki) for i = 1;n
true ➔ [ide-1/col-1,…,ide-n/col-n]

Of course, each of these function is an inverse of the other, which means that the following functions are identities:

C2R ● R2C
R2C ● C2R

11.2.5 Databases and their subordination relations

By a *database* we shall mean a mapping assigning tables to their names:

dab : DatBas = Identifier ⇒ Table.

Databases, similarly to tables, belong to the category of values since they combine data and types. The types of databases are implicit in the types of their tables and carry two kinds of information:

- pieces of information about individual tables, saved in the types of columns (except column markings) and in row yokes,
- information about subordination relationships between tables, saved in column markings.

To define the relationships between tables, consider a database dab, two identifiers ch-ide and pa-ide which are the names of two tables in this base and an identifier co-ide which is a common name of two columns of these tables. Let further:

ch-tab = dab.ch-ide child table
(ch-ctc, (ch-tah, ch-roy)) = ch-tab
ch-col = ch-ctc.co-ide child column
(ch-bat, ch-bav, ch-coy, ch-com) = ch-tah.co-ide child-column type

pa-tab = dab.pa-ide parent table
(pa-ctc, (pa-tah, pa-roy)) = pa-tab parent column
pa-col = pa-ctc.co-ide
(pa-bat, pa-bav, pa-coy, pa-com) = pa-tah.co-ide parent-column type

We say that a *content-subordination relation* (OWN) holds between our tables via column name co-ide that we shall write as

ch-ide <u>content-subordinated</u> [co-ide] pa-ide,

if two following conditions are satisfied:

1. the *child column* ch-col contains only such elements that are included in the parent column,
2. the *parent column* pa-col is repetition-free.

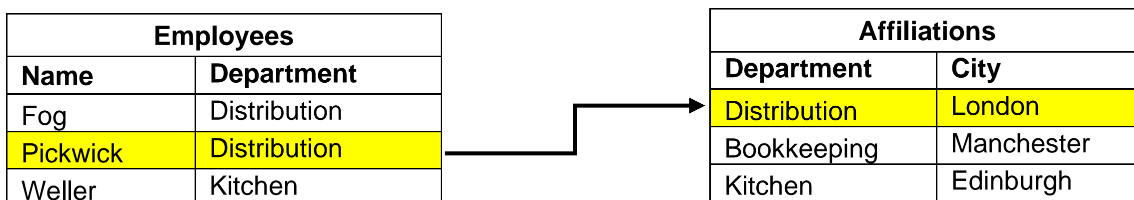
In such a case we shall say that:

- ch-tab is a *child table*, pa-dab is its *parent table* and co-ide is their *subordination connector,*
- ch-tab.co-ide is a *child column,* pa-tab.co-ide is its *parent column,*

The conjunction of 1. and 2. implies that each row of the child table unambiguously identifies — *points to* — a row in the parent table. Why we formally define the subordination between the names of tables rather than between the tables themselves will be seen in a moment.

In Fig. 11.2-1, we see an example of two tables (we show only their contents) which are in a content-subordination relation:

Employees <u>content-subordinated</u> [Department] Affiliations



Employees	
Name	**Department**
Fog	Distribution
Pickwick	Distribution
Weller	Kitchen

Affiliations	
Department	**City**
Distribution	London
Bookkeeping	Manchester
Kitchen	Edinburgh

Fig. 11.2-1 Employees **is a child of** Affiliations **via** Department

Notice that there may be some elements in a parent column that do not appear in the corresponding child column, e.g., departments with no employees. It is also possible that a child column for one table is, at the same time, a parent or a child column for another table, or that it is a child column for more than one table. However, we exclude cases where a table is a child of itself, although cycles are allowed. And, of course, one parent may have many children.

Note now that an existing subordination relation between two tables may be destroyed when we modify one of these tables, e.g. if to a child column we add a value that does not appear in the parent column, or if we remove or change a value in a parent column. To prevent such situations whenever a subordination is critical, SQL offers a column marking mechanism, allowing programmers to mark a column as a child or a potential parent.

We have seen examples of such markings in Sec. 10.3 where:

- child table Employees includes column Department_ID marked as a child by REFERENCES Affiliations,
- parent table Affiliations includes column Department_ID marked as a parent by PRIMARY KEY.

To include this mechanism in our model we introduce a second relation between tables called a *marking-subordination relation* written as

ch-ide <u>marked-subordinated</u> [ide] pa-ide,

which holds if:

- pa-ide : ch-com; in this case we say that:
  - column ch-tab.co-ide is *marked to be a child* of parent column pa-tab.co-ide,
  - table ch-tab is *marked to be a child* of parent table pa-tab,
- 'primary' : pa-mar; in this case we say that:
  - column pa-tab.co-ide is *marked to be a parent column,*
  - table pa-tab is *marked to be a parent table.*

Note that child marking specifies parents, whereas parent marking does not specify children. Two tables are said to be in a *subordination relation via* co-ide, in symbols

ch-ide <u>subordinated</u> [co-ide] pa-ide

if both corresponding relations hold, i.e., if

- ch-ide <u>content-subordinated</u> [co-ide] pa-ide, and
- ch-ide <u>marked-subordinated</u> [co-ide] pa-ide.

A database is said to be *consistent* (OWN) if all tables marked as subordinated are content subordinated, although not necessarily vice-versa. Contents of two tables may "happen to be subordinated," but if they are not marked as subordinated, the SQL engine does not need to protect their subordination. In the opposite case potential consistency violation must be prevented either by signalizing an error and aborting program execution or by an appropriate modifications of tables. These mechanisms will be described in Sec. 11.3.4.

Since the types of tables in a database describe user-defined properties of the database, they are called *integrity constraints* of the database.

If a column marking of a column named co-ide includes a parent-table name pa-ide then the pair of identifiers (co-ide, pa-ide) is called a *coupling pair*.

The following operations on tables may violate database consistency:

1. when we modify a column marked as a <u>parent</u>:
   a. a <u>violation of repetition-freeness</u> of the column; this situation may happen if we:
      i. add a new row to the table, i.e., add a new value to the column,
      ii. replace a value in the column by another value,
   b. a <u>violation of parent adequacy</u> of the column; this situation may happen if we:
      i. remove a row from the table, i.e., remove a value from the column,
      ii. replace a value in the column by another value,
2. when we modify a column marked as a <u>child</u>:
   a. a <u>violation of child adequacy</u> of the column; this situation may happen if we:
      i. add such a row to the table that the value added to the column does not appear in the parent column,
      ii. replace a value in the column by a value that does not appear in the parent.

In the sequel, all operations on tables will be defined in such a way that the consistency of a databases will be secured either by aborting a consistency-braking action or by initiating a cascade of recovery actions. The following auxiliary function will be used to check if the subordination relation holds in a database between three identifiers. This function returns either a truth value tv or an error message that indicates why the relation does not hold.

```
subordination : DatBas x Identifier x Identifier x Identifier ⟼ {‘OK’} | Error
subordination.(dab, ch-ide, pa-ide, co-ide) =
  dab.ch-ide = ?                            ➔ ‘no child table in the base’
  dab.pa-ide = ?                            ➔ ‘no parent table in the base’
  let                                                              for i = 1;n
    (ch-ctc, ch-tat)  = dab.ch-ide
    (ch-tah, ch-roy) = ch-tat
    (pa-ctc, pa-tat)  = dab.pa-ide
    (pa-tah, pa-roy) = pa-tat
  ch-tah.co-ide = ?                         ➔ ‘no such column in child’
  pa-tah.co-ide = ?                         ➔ ‘no such column in parent’
  let
    ch-col                                  = ch-ctc.co-ide       child column
    (ch-bat, ch-bav, ch-coy, ch-com)        = ch-tah.co-ide
    pa-col                                  = pa-ctc.co-ide       parent column
    (pa-bat, pa-bav, pa-coy, pa-com)        = pa-tah.co-ide
  ‘primary’  /: pa-com                      ➔ ‘parent not marked’
  pa-ide     /: ch-com                      ➔ ‘parent not expected’
  are-reperitions.pa-col                    ➔ ‘repetitions in parent column’
```

```
    elements.ch-col /⊆ elements.pa-col   ➔ 'missing elements in parent column'
    true                                  ➔ 'OK'
```

Of course

ch-ide <u>subordination</u> [co-ide] pa-ide holds in dba iff
subordination.(dab, co-ide, ch-tab, pa-tab) = 'OK'.

One of typical cases when a database is not consistent is when some of its child-marked tables are orphans.

A column named co-ide is a *column orphan* of a parent table named pa-ide if it is marked as a child of pa-ide but content-wise it is not its child. A table is said to be a *table orphan* of another table if it includes a column orphan of that other table. Note that only children may be orphans.

The following function checks if a given column is an orphan in a given table, and if that is the case, it describes the cause. Precisely speaking, it first checks if the given column is marked as a child and, if that is the case — if it is an orphan of one of its indicated parents. Note that this function returns 'OK' if the tested column <u>is not</u> an orphan. This "reaction" seems adequate, since we do not want orphans in our databases.

```
col-is-orphan : DatBas x Identifier x Identifier ↦ {'OK'} | Error
col-is-orphan.(dab, ta-ide, co-ide) =
    dab.ta-ide = ?                         ➔ 'no such table'
    let
        (ctc, tat)   = dab.ta-ide
        (tah, roy)   = tat
    tah.co-ide = ?                         ➔ 'no such column'
    let
        (bat, bav, coy, com)  = tah.co-ide
        ch-col                = ctc.co-ide
        parents               = com – {'primary'}                      the set of parents' names
    parents = {}                           ➔ 'OK'
    let
        {pa-ide-1,…,pa-ide-n}   = parents
    dab.pa-ide-i = ?                       ➔ 'parent not in the base'       for i = 1;n
    let
        (pa-ctc-i, pa-tat-i) = dab.pa-ide-i                                 for i = 1;n
    pa-ctc-i.co-ide = ?                    ➔ 'column not in the parent'     for i = 1;n
    let
        pa-col-i = pa-ctc-i.co-ide                                          for i = 1;n
    elements.ch-col /⊆ elements.pa-col-i   ➔ 'inadequate parent column'     for i = 1;n
    true                                   ➔ 'OK'                           column is not an orphan
```

As we see, a column may be an orphan if it is a child and:

- its expected parent table does not exist in the database,
- its expected parent exists, but the expected column does not exist in it,
- the expected column exists, but is not adequate.

The following predicate checks if a table is an orphan of one of its parents:

```
tab-is-orphan : DatBas x Identifier ↦ {'OK'} | Error
tab-is-orphan.(dab, ta-ide) =
    dab.ta-ide = ?                         ➔ 'no such table'
    let
        tab             = dab.ta-ide
        (ctc, tat)      = tab
        {ide-1,…,ide-n} = dom.ctc
```

```
    col-is-orphan.(dab, ta-ide, ide-i) : Error  ➔ col-is-orphan.(dab, ta-ide, ide-i)   for i = 1;n
    true                                        ➔ 'OK'                           table is not an orphan
```

Subsequent predicate checks if for a column marked as a parent there are its children that are orphans:

```
col-has-orphan : DatBas x Identifier x Identifier ⟼ {'OK'} | Error
col-has-orphan.(dab, pa-ide, co-ide) =
    dab.pa-ide = ?                                       ➔ 'no such table'
    let
      {ide-1,…,ide-n}         = dom.dab                  the names of all tables in the current base
      (ctc-i, (tah-i, roy-i))  = dab.ide-i   for i = 1;n
      (ctc, (tah, roy))        = dab.pa-ide
    tah.co-ide = ?                                       ➔ 'no such column'
    'primary' /: marking.(tah.co-ide)                    ➔ 'column is not a parent'
    are-repetitions.(ctc.co-ide)                         ➔ 'parent not repetition-free'
    (∃i) (ide-i : marking.(tah.co-ide)  and
             ctc-i.co-ide = !           and
             elements.(ctc-i.co-ide) /⊆ elements.(ctc.co-ide)) ➔ 'orphan exists'
    true                                                 ➔ 'OK'              no orphans
```

11.2.6 States

Intuitively, the enrichment of **Lingua** to **Lingua-SQL** consists in adding some new mechanisms to our language, whereas all former mechanisms are retained. There seem to be two alternative solutions to achieve this goal:

1. an enlargement of the domains of values, types and references by new elements whereas the structure of states remain unchanged,
2. an expansion of the structure of states by SQL components carrying storable SQL entities.

We chose the second solution since it seems more convenient to show, on the one hand, the borderline between the source language and its SQL part and, on the other — an interface between these two components. We assume, therefore, that our *new states* will be pairs:

```
nes : NewSta = State x SqlSta
```

consisting of a former state, which we shall call a *hereditary component,* and a *SQL component* that will carry tables, databases and transactions. We take the following introductory assumptions about our SQL model:

1. Time-and-date data, types and typed data will be included into **Lingua** domains of simple data, types and typed data respectively. Consequently variables pointing to these typed data will be stored in hereditary components of states.
2. SQL types and yokes will not be storable.
3. We do not introduce a covering relation between SQL types; types compatibility simply means that the involved types must be equal.
4. We shall not introduce expressions that generate tables and databases. Both will be created by declaration and then assigned to appropriate variables.
5. In SQL stores, tables and databases, will be assigned to variables directly, rather than via references. We do not introduce references in the SQL part of our model since objects will not carry neither tables nor databases.
6. Tables and databases will be public.

The domain of SQL states is defined as follows:

```
sta   : SqlSta      = TraEnv x SqlSto                        SQL states
```

tre : TraEnv = Identifier ⇒ Transaction transaction environments
sto : SqlSto = Database x DbaRep x Monitor SQL stores
dbs : DbaRep = Identifier ⇒ Database database repository
mon : Monitor = Identifier | {Ω} monitors

Transactions are blocks of table-modifying instructions and will be described in Sec. 11.3.4.3.

The (unique) database that is a component of an SQL store is the *currently active database*, and its tables are called *active tables*. We assume that only one database may be active at a time. Database repository binds databases that are available but currently *not active*.

The monitor is either a name of an *active table,* which means that this table is displayed on a computer monitor, or is Ω, which means that the monitor does not display anything. A typical new state is, therefore, of the following form:

(((cle, tye, cov), (obn, dep, ota, sft, err)), (tre, (dab, dbr, mon)))

A new state is said to be *well-formed*, if:

1. its hereditary component is well-formed (Sec. 5.3),
2. every identifier appearing in a new state, appears in it only once,
3. the active database and all databases in the repository are consistent.

The set of well-formed new states will be denoted by

WfNewSta

Functions is-error, error, ◄ and declared (Sec. 5.3) are extended to new states in an obvious way.

11.3 The algebra of denotations

11.3.1 Replicated denotations and their constructors

To incorporate in our model the rule that all the mechanism of **Lingua** are retained in **Lingua-SQL**, we introduce the concept of a *replicated denotation*. Given a state-dependent denotation of **Lingua**, e.g., an imperative denotation

den : WfState → WfState

by the *replica* of den, we mean the following function on new states:

R[den] : WfNewState → WfNewState
R[den].(sta, sql-sta) =
  den.sta = ? ➔ ?
  true ➔ (den.sta, sql-sta)

A denotation is said to be a *replicated denotation* if it is a replica of a **Lingua** denotation. Replicated applicative denotations, i.e., the denotations of expressions, are defined in an analogous way.

Replicas of state-independent denotations, i.e., of primitive denotations (cf. Sec. 6.1) and yoke-expression denotations, are just these denotations.

Now, with every domain Den of **Lingua** denotations we associate a corresponding domain of replicated denotations:

R[Den] = {R[den] | den : Den}

Carrier R[Den] is called a replica of the source carrier Den. Further on, with every **Lingua** constructor of denotations

cons : Den-1 x … x Den-n ⟼ Den

we associate a *replica of this constructor* defined in the following way:

R[cons] : R[Den-1] x … x R[Den-n] ⟼ R[Den]
R[cons].(R[den-1],…,R[den-n]) = R[cons.(den-1,…,den-n)]

Note that this definition is correct, since R is a 1-1 function. The following example illustrates the idea of replication:

while : ValExpDen x InsDen ⟼ InsDen

R[while] : R[ValExpDen] x R[InsDen] ⟼ R[InsDen]

R[while].(R[ved], R[ins]) = R[while.(ved, ind)] i.e.

R[while].(R[ved], R[ins]).(sta, sql-sta) =
while.(ved, ins).sta = ? ➔ ?
true ➔ (while.(ved, ins).sta, sql-sta)

In Sec. 6.1, by **AlgDen** we have denoted the algebra of denotations of **Lingua**. Let **AlgDenSQL** denote our future algebra of **Lingua-SQL** denotations. We assume that all constructors of **AlgDen** will have their corresponding replicas in **AlgDenSQL** which implies that all reachable denotations of **AlgDen** will have their replicas in **AlgDenSQL**. The remaining denotations and their constructors will correspond to SQL mechanisms, and will be described in the subsequent sections.

11.3.2 The carriers of the algebra of denotations

In our new algebra of denotations **AlgDenSQL** we are going to have three categories of carriers:

1. all primitive carriers of the former algebra plus two new primitive carriers of check settings and column markings,
2. the replicas of all carriers of state-dependent denotations of the former algebra,
3. new SQL carriers.

Metavariables running over replicated domains will be denoted by the same symbols as for the corresponding original domains.

Primitive carriers

ide : Identifier = … identifiers
prs : PriSta = {'private', 'public'} privacy statuses indicators
loi : ListOfIde = Identifier^{c*} lists of identifiers
cli : ClaInd = {'empty-class'} | Identifier class indicators
chs : CheSetting = {'restrict', 'cascade'} check settings
com : ColMar = FinSub.(Identifier | {'primary'}) column markings

Applicative carriers

yok : R[YokExpDen] = Yoke yoke-expression denotations
ted : R[TypExpDen] = WfNewSta ⟼ TypeE replicated type-expression denotations

ved : R[ValExpDen] = WfNewSta → ValueE replicated value-expression denotations
btd : BasTypExpDen = WfNewSta → BasTypE basic-type expression denotations
red : RowExpDen = WfNewSta → LabRowE row expression denotations
ryd : RowYokExpDen = WfNewSta → RowYokE row-yoke expression denotations
cyd : ColYokExpDen = WfNewSta → ColYokE column-yoke expression denotations
cmd : ColMarExpDen = ColMar column-marking expression denotations
ctd : ColTypExpDen = WfNewSta → ColTyp column-type expression denotations
thd : TabHeaExpDen = WfNewSta → TabHeaE table-header expression denotations
ttd : TabTypExpDen = WfNewSta → TabTypE table-type expression denotations

Imperative carriers

dcd : R[DecDen] = WfNewSta → WfNewSta replicated declaration denotations
pod : R[ProOpeDen] = WfNewSta ⟼ WfNewSta replicated procedure opening denotation
ctc : R[ClaTraDen] = Identifier ⟼ WfNewSta → WfNewSta repl. class-trans. denotations
ind : R[InsDen] = WfNewSta → WfNewSta replicated instruction denotations
ppd : R[ProPreDen] = WfNewSta → WfNewSta replicated program preamble denotations
prd : R[ProDen] = WfNewSta → WfNewSta replicated program denotations
tdd : TabDecDen = WfNewSta ⟼ WfNewSta table-declaration denotation
nid : NewInsDen = WfNewSta → WfNewSta new instruction denotations

Declaration-oriented carriers

dse : R[DecSec] = ListOfIde x R[TypExpDen] new declaration sections
fpd : R[ForParDen] = $R[DecSec]^{c*}$ new formal-parameter-denotations
apd : ActParDen = ListOfIde new actual-parameter-denotations

Signature carriers

ips : R[ImpProSigDen] = R[ForParDen] x R[ForParDen] new imperative-procedure sig. den.
fps : R[FunProSigDen] = R[ForParDen] x R[TypExpDen] new functional-procedure sig. den.
ocs : R[ObjConSigDen] = R[ForParDen] x Identifier new object-constructor signature den.

The constructors of our new algebra include replicas of all former constructors, plus some new constructors that we shall call *SQL constructors*. In the following sections we shall define a few most typical examples of SQL constructors.

11.3.1 Constructors of primitive denotations

The only new constructors in this category are constructors of column markings. We shall need two such constructors:

```
com-create-ma.mar  :                    ⟼ ColMar                for every mar : Mark
com-add-ma         : ColMar x Mark ⟼ ColMarE
```

Where

```
mar : Mark = Identifier | {'primary'}
```

We skip obvious definitions of our constructors.

11.3.2 Expressions

11.3.2.1 Categories of SQL expressions

Whereas in the case of **Lingua** we had only three categories of expressions — value expressions, type expressions and yoke expressions — in **Lingua-SQL** we add eight new categories:

1. basic-type expressions,
2. row expressions,
3. row-yoke expressions,
4. column-yoke expressions,
5. column-marking expressions,
6. column-type expressions,
7. table-header expressions,
8. table-type expressions.

11.3.2.2 Basic-type expressions

Although we have assumed (Sec. 11.2.6) that basic types are not storable (an engineering decision) their denotations must be functions on states, since decimal- and character types include integers, that we have to "somehow generate". For this sake we shall use value expressions[106].

```
btd : BasTypExpDen = WfNewSta → BasTypE
```

Integers needed in building types are computed by means of the replicas of value expressions, e.g.,

```
btd-create-decimal : R[ValExpDen] x R[ValExpDen] ⟼ BasTypExpDen           i.e.
btd-create-decimal : R[ValExpDen] x R[ValExpDen] ⟼ WfNewSta → BasTypE
btd-create-decimal.ved-1.ved-2.nes =
   is-error.nes                 ➔ error.nes
   ved-i.nes = ?                ➔ ?                        for i = 1,2
   let
      val-i = ved-i.nes                                    for i = 1,2
   val-i : Error                ➔ val-i                    for i = 1,2
   sort-t.val-i ≠ 'integer'     ➔ 'integer expected'       for i = 1,2
```

[106] This solution is, maybe, a little "too far going" since we essentially do not need the "whole power" of value expressions to calculate constants indicating the sizes of decimal and character types. However, an alternative solution would be introducing a particular category of expressions generating these constants, thus complicating our model even more.

val-i < 0 **or** val-i > max-i ➔ 'parameter out of scope' for i = 1,2
true ➔ ('De', val-1, val-2)

where max-i's are parameter indicating maximal numbers in decimal types. The definitions of remaining constructors of the category are analogous.

11.3.2.3 Row expressions

The denotations of row expressions, given states return labelled rows:

red : RowExpDen = WfNewSta ⟼ LabRowE

The first constructor of these denotations creates a denotation that given a state creates a one-element row with empty field:

red-build-empty-field-row : Identifier ⟼ RowExpDen i.e.
red-build-empty-field-row : Identifier ⟼ WfNewSta → LabRowE
red-build-empty-field-row.ide.nes = [ide/Ω]

Next constructors builds a denotation that creates a one-element row data with a non-empty field:

red-build-nonempty-field-row : Identifier x R[ValExpDen] ⟼ RowExpDen i.e.
red-build-nonempty-field-row : Identifier x R[ValExpDen] ⟼ WfNewSta → LabRowE
red-build-nonempty-field-row.ide.ved.nes =
is-error.nes ➔ error.nes
ved.nes = ? ➔ ?
ved.nes : Error ➔ ved.nes
let
val = ved.nes
sort-t.val /: BasTyp ➔ 'basic-type expected'
true ➔ [ide/val]

The third constructor corresponds to adding an empty field to a row data:

red-add-empty-field : Identifier x RowExpDen ⟼ RowExpDen i.e.
red-add-empty-field : Identifier x RowExpDen ⟼ WfNewSta → LabRowE
red-add-empty-field.ide.red.nes =
is-error.nes ➔ error.nes
red.nes = ? ➔ ?
red.nes : Error ➔ red.nes
let
lar = red.nes
red.ide = ! ➔ 'identifier already bound'
true ➔ red[ide/Ω]

The last constructor corresponds to adding a nonempty field to a row data:

red-add-nonempty-field : Identifier x R[ValExpDen] x RowExpDen ⟼ RowExpDen

We skip its obvious definition.

11.3.2.4 Column-yoke expressions

Column-yoke expressions generate column yokes and therefore their denotations constitute the following domain:

cyd : ColYokExpDen = WfNewSta → ColYokE

Similarly as in Sec. 6.4.2 the constructors of column-yoke expression denotations are derived from corresponding yoke constructors. We give two examples to illustrate this idea.

cyd-qcy-greater-in : R[ValExpDen] ⟼ ColYokExpDen i.e.
cyd-qcy-greater-in : R[ValExpDen] ⟼ WfNewSta → ColYokE

```
cyd-qcy-greater-in.ved.nes =
   is-error.nes  ➔ error.nes
   ved.nes = ?   ➔ ?
   let
      val = ved.nes
   val : Error   ➔ val
   true          ➔ qcy-greater-in.val
```

This constructor builds a denotation that generates a quantified yoke that checks if all elements of a column are greater than a given integer. Next constructor builds a denotation that generates a holistic yoke that check if a column has repetitions

```
cyd-hcy-unique : ↦ ColYokExpDen
cyd-hcy-unique : ↦ WfNewSta → ColYokE
cyd-hcy-unique.().nes = hcy-unique.()
```

11.3.2.5 Row-yoke expressions

Analogously to the former expressions *row-yoke expressions* evaluate to row yokes:

```
ryd   : RowYokExpDen  = WfNewSta → RowYokE
```

The following constructor builds a denotation that generates a row yoke that adds two integers generated by two given row yokes:

```
ryd-add-int : RowYokExpDen x RowYokExpDen ↦ RowYokExpDen                i.e.
ryd-add-int : RowYokExpDen x RowYokExpDen ↦ WfNewSta → RowYokE
ryd-add-int.(ryd-1, ryd-2).nes =
   is-error.nes : Error   ➔ error.nes
   ryd-i.nes = ?          ➔ ?               for i = 1,2
   ryd-i.nes : Error      ➔ ryd-i.nes       for i = 1,2
   let
      roy-i  = ryd-i.nes                    for i = 1,2
   true                   ➔ ry-add-in.(roy-1, roy-2)
```

11.3.2.6 Column-marking expressions

Column-marking expressions are state-independent and therefore their denotations are just column markings

```
cmd : ColMarExpDen = ColMar
```

and their constructors are column-marking constructors defined in Sec. 11.2.2.

11.3.2.7 Column-type expressions

Column-type expressions evaluate to column types:

```
ctd : ColTypExpDen = WfNewSta → ColTypE
```

We need only one constructor to build their denotations:

```
ctd-create : BasTypExpDen x ValExpDen x ColYokExpDen x ColMarExpDen ↦ CalTypExpDen
ctd-create : BasTypExpDen x ValExpDen x ColYokExpDen x ColMarExpDen ↦
                                                             WfNewSta → ColTypE
ctd-create.(btd, ved, cyd, cmd).nes =
   is-error.nes     ➔ error.nes
   btd.nes  = ?     ➔ ?
   ved.nes = ?      ➔ ?
   cyd.nes = ?      ➔ ?
   let
      bat    = btd.nes
```

```
      val    = ved.nes
      cyd    = cyd.nes
   bat : Error          ➔ bat
   val : Error          ➔ val
   cyd : Error          ➔ cyd
   sort-t.val ≠ bat     ➔ 'inconsistent types'
   true                 ➔ (bat, val, cyd, com)
```

11.3.2.8 Table-header expressions

Table-header expressions evaluate to table headers (Sec. 11.2.4):

thd : TabHeaExpDen = WfNewSta → TabHeaE

We need two constructors to build them

```
thd-create        : Identifier x ColTypExpDen                  ⟼ TabHeaExpDen
thd-add-column    : Identifier x ColTypExpDen x TabHeaExpDen   ⟼ TabHeaExpDen
```

We skip their obvious definitions.

11.3.2.9 Table-type expressions

Table-type expressions evaluate to table types:

ttd : TabTypExpDen = WfNewSta → TabTypE

They are built by one constructor:

ttd-create : TabHeaExpDen x RowYokExpDen ⟼ TabTypExpDen

Its definition is obvious.

11.3.3 Declarations of table variables

A declaration of a table variable builds a one-row table with default values and assigns this table to an identifier in the active database. If the declared table expects parents then the declaration checks if such parents exist in the base. A consequence of this rule is that parents must be declared and filled with values prior to the declarations of their children (which makes sense biologically as well!).

```
declare-table : Identifier x TabTypExpDen ⟼ TabDecDen                i.e.
declare-table : Identifier x TabTypExpDen ⟼ WfNewSta → WfNewSta
declare-table.(ta-ide, ttd).nes =
   is-error.nes                              ➔ nes
   declared.ta-ide.nes                       ➔ nes ◄ 'variable already declared'
   ttd.nes = ?                               ➔ ?
   ttd.nes : Error                           ➔ ttd.nes
   let
      (sta, (tre, (dab, dbr, mon)))        = nes
      tat                                  = ttd.nes
      ([ide-1/cot-1,…,ide-n/cot-n], roy)   = tat
      (bat-i, bav-i, cyd-i, com-i)         = cot-i        for i = 1;n        column types
      ctc                                  = [ide-1/bav-1,….,ide-n/bav-n]    column table-content
      tab                                  = (ctc, tat)
      new-dab                              = dab[ta-ide/tab]
   tab-is-orphan.(new-dab, ta-ide) : Error   ➔ nes ◄ tab-is-orphan.(dab, ta-ide)
   true                                      ➔ (sta, (tre, (new-dab, dbr, mon)))
```

The inspection of the definition of col-is-orphan.(dab, ta-ide, ide-i) in Sec. 11.2.5 shows that it generates an error only if ta-ide is not in dab, which in our case can't happen. Note also that the declared table is conformant with its type.

11.3.4 Instructions

11.3.4.1 Row-oriented table instructions

The first constructor adds a row to a declared table. The new row is generated as a new labelled row by a row expression, and then every value in this row is added to a corresponding column. Two categories of type checks are necessary in this operation:

1. checking if the new table does not violates its type integrity,
2. checking if the new table does not violate the database consistency, which may happen if we add a new element to a child column which makes the new column an orphan.

Add a row to a table

```
nid-add-ro-to-ta : Identifier x RowExpDen ↦ NewInsDen                i.e.
nid-add-ro-to-ta : Identifier x RowExpDen ↦ WfNewSta → WfNewSta
nid-add-ro-to-ta.(ta-ide, red).nes =
   is-error.nes                              ➔ nes
   let
      (sta, (tre, (dab, dbr, mon)))   = nes
   dab.ta-ide = ?                            ➔ 'no such table'
   red.nes = ?                               ➔ ?
   red.nes : Error                           ➔ nes ◄ red.nes
   let
      tab                          = dab.ta-ide
      (ctc, tat)                   = tab
      [ide-1/col-1,…,ide-n/col-n]  = ctc                table-content to be modified
      lar                          = red.nes            labeled row to be added
   dom.lar ≠ {ide-1,…,ide-n}                 ➔ 'incorrect row identifiers'
   let
      new-val-i       = lar.ide-i                      for i = 1;n
      new-col-i       = col-i @ (new-val-i)            for i = 1;n
      new-ctc         = [ide-1/new-col-1,…,ide-n/new-col-n]     new column table-content
      tab-message     = type-table-check.(new-ctc, tat)          (Sec. 11.2.4)
   tab-message : Error                       ➔ nes ◄ tab-message
   let
      ([ide-1/cot-1,…,ide-n/cot-n], roy)   = tah
      (bat-i, bav-i, coy-i, com-i)         = cot-i             for i = 1;n
      new-tab                              = (new-ctc, tat)
   col-is-orphan.(dab, ta-ide, ide-i) : Error  ➔ nes ◄ col-is-orphan.(dab, ta-ide, ide-i)
   true                                      ➔ (sta, (tre, (dab[ide/new-tab], dbr, mon)) )
```

Note that if at any stage of our table modification an error message is raised, then it is loaded to the error register of the initial state, which means that the initial database remains unchanged and the intended modification is abandoned.

All constructors that follow will be defined according to the following scheme:

1. the transformation of a column content of a table into a row content,

2. a modification of the row content by an appropriate constructor of labelled rows,
3. the transformation of the resulting row content into a new column content,
4. checking integrity constraints of the new table,
5. checking integrity constraints of the new database and possibly initiating a cascade action.

Remove from a table all rows that satisfy a given yoke

```
nid-cut-ro-from-ta : RowYokExpDen x Identifier x CheSetting ⟼ NewInsDen            i.e.
nid-cut-ro-from-ta : RowYokExpDen x Identifier x CheSetting ⟼ WfNewSta → WfNewSta
nid-cut-ro-from-ta.(ryd, ta-ide, chs) =
   is-error.nes                                 ➔ nes
   let
      (sta, (tre, (dab, dbr, mon)) ) = nes
   dab.ta-ide = ?                               ➔ 'no such table'
   ryd.nes = ?                                  ➔ ?
   ryd.nes : Error                              ➔ ryd.nes
   let
      roy              = ryd.nes
      (ctc, tat)       = dab.ide
      (lar-1,…,lar-k)  = C2R.ctc
   k = 1                                        ➔ nes ◄ 'single row can't be removed'
   let
      rtc              = drop-rows.(roy, (lar-1,…,lar-k))     row-table-content of the new table
      new-ctc          = R2C.rtc                              new column table-content
      tab-message      = type-table-check.(new-ctc, tat)
   tab-message : Error                          ➔ nes ◄ tab-message
   let
      new-tab                        = (new-ctc, tat)
      new-dab                        = dab[ta-ide/new-tab]
      [ide-1/col-1,…,ide-n/col-n]    = new-ctc
      (tah, roy)                     = tat
      [ide-1/cot-1,…,ide-n/cot-n]    = tah
      (bat-i, bav-i, coy-i, com-i)   = cot-i       for i = 1;n
   each of new parent columns may have orphans
   col-has-orphan.(new-dab, ta-ide, ide-i) ≠ 'OK' and                for i = 1;n
   col-has-orphan.(new-dab, ta-ide, ide-i) ≠ 'column is not a parent'➔
      chs = 'restrict'      ➔ nes ◄ col-has-orphan.(new-dab, ta-ide, ide-i)
      chs = 'cascade'       ➔ … cascading removal of orphans in the table
   true                                         ➔ (sta, (tre, (new-dab, dbr, mon)) )
```

This constructor first transforms the given column content of the table into a row content. From this content it removes all labelled rows that satisfy the yoke. This action is performed by using an auxiliary function drop-rows; we skip its formal definition. The resulting row content of the table is transformed (back) into a column table-content. After this modification of our table we have to perform two checks.

In the first place we check if the new table content is of the former table type. Note that although the dropping of column elements won't spoil quantified yokes it may spoil holistic ones (Sec. 11.2.2). If this check is negative we simply generate an error message and the table remains unchanged.

In the second checking step, we check if removing some elements from parent-marked columns of our table did not cause them to have orphans. To do that we use function col-has-orphan and we check column-by-column the new table. Once we find a column which is a parent and has orphans, we either abort our instruction and report an error message —that in this case will be 'orphan exists' — or we initiate a cascade action of removing orphans, and their orphans, and their orphans… .

E.g. (cf. Fig. 11.2-1), if the Distribution row is removed from Affiliations, then the column Department_ID in Employees becomes an orphan. If this happens, the further action depends on the check-setting chs.

- If it is set to 'restrict', then the operation is abandoned and an error is signalized.
- If it is set to 'cascade', then we remove from Employees all rows that include Distribution.

We shall not formalize the "cascade" part of our definition, since it is a technical task which would not contribute much to our model.

Remove all rows of the first table from the second table

nid-exclude-ro-from-ta : Identifier x Identifier x CheSetting ⟼ NewInsDen i.e.
nid-exclude-ro-from-ta : Identifier x Identifier x CheSetting ⟼ WfNewSta → WfNewSta

The definition of this constructor is analogous to the former, we just have to use another auxiliary function on tuples.

11.3.4.2 Column-oriented table instructions

The four constructors to be defined in this section are associated with columns only implicitly since none of them neither takes columns as arguments nor return them as a results. All of them are defined in five steps analogous to these of row-oriented constructors:

1. the transformation of a column table-content into a row table-content,
2. a modification of every labeled row by an appropriate constructor of labeled rows,
3. the transformation of the resulting row-table-content into a new table (in the cases of constructors add, cut, change) or into a column (in the case of constructor get).
4. checking integrity constraints of the new table,
5. checking integrity constraints of the new database.

Add a column to a table

nid-add-column : Identifier x Identifier x ColTypExpDen ⟼ NewInsDen
nid-add-column : Identifier x Identifier x ColTypExpDen ⟼ WfNewSta → WfNewSta
nid-add-column.(ta-ide, co-ide, ctd).nes = ta – table, co- column
is-error.nes ➔ nes
ctd.nes = ? ➔ ?
ctd.nes : Error ➔ ctd.nes
let
cot = ctd.nes
(sta, (tre, (dab, dbr, mon))) = nes
dab.ta-ide = ? ➔ nes ◄ 'no such table'
let
(ctc, tat) = dab.ta-ide table to be modified
ctc.co-ide = ! ➔ nes ◄ 'column already exists'
let
n = depth.ctc
(bat, bav, coy, com) = cot
col = bav^{cn} column of default values of length n
col-message = check-column-type.(col, cot)
col-message ≠ 'OK' ➔ nes ◄ col-message
let
new-ctc = ctc[co-ide/col] new column table-content
(tah, roy) = tat source table type

new-hea = tah[co-ide/cot] new table header
new-tat = (new-hea, roy) new table type
new-tab = (new-ctc, new-tat) new table
new-dab = dab[ta-ide/new-tab] new database
here we start the consistency check of the new database
com = {} ➔ (sta, (tre, (new-dab, dbr, mon)))
'primary' : com ➔ nes ◄ 'repetitions in a parent column' (since all elem. are the same)
let
{ide-1,…,ide-k} = com
result-i = subordination.(new-dab, ta-ide, ide-i, co-ide) for i = 1;k
result-i : Error ➔ nes ◄ result-i for i = 1;k
true ➔ (sta, (tre, (new-dab, dbr, mon))

This constructor first builds a new column of (identical) default values indicated by the type of this column, and then it checks if the new column satisfies the assumed column type. If that is the case, the table type is augmented by the new column type but its row yoke remains unchanged. If we want to cover new column by the row yoke, we have to modify the latter by a dedicated instruction.

In the end we perform the consistency check:

- if the new column is neither a marked parent not a marked child, then we are done,
- otherwise, if the column is marked as a parent, then we signalize an error, since our column includes repetitions; all its elements are equal to the default value,
- otherwise we identify all tables indicated as parents of our column (and table), and we check if they are actually parents.

Cut a column from a table

The simple removal of a column from a table is an easy task, and therefore the bulk of the work is in checking if the resulting database satisfies all integrity constraints. There are essentially two cases when the removal of a column may destroy the integrity of a base:

1. A removal of a column from a table means a removal of an element from each row, and this may cause the row yoke of the table to be not satisfied anymore.
2. If the column to be removed is marked as a parent, then its removal may cause orphanhood of some tables in the base. In this case the actions to be taken depends on the setting the checking parameter:
    a. if it is set to 'restrict', then an error message is generated and further action is aborted,
    b. if it is set to 'cascade', then the instruction removes all these children columns from children tables that violate database consistency; during this action it permanently check if the row yokes of the modified tables are satisfied, and if they are not, it aborts the cascade and returns to the initial state.

nid-cut-column : Identifier x Identifier x CheSetting ⟼ NewInsDen
nid-cut-column : Identifier x Identifier x CheSetting ⟼ WfNewSta → WfNewSta
nid-cut-column.(ta-ide, co-ide, chs).nes = ta – table, co- column
is-error.nes ➔ nes
let
(sta, (tre, (dab, dbr, mon))) = nes
dab.ta-ide = ? ➔ nes ◄ 'no such table'
let
(ctc, tat) = dab.ta-ide

```
    ctc.co-ide = ?          ➔ nes ◄ 'no such column'
    let
        (tah, roy)              = tat                          source table type
        (bat, bav, coy, com)    = tah.co-ide
        new-ctc                 = ctc[co-ide/?]                new column table-content
        (row-1,…,row-n)         = C2R.new-ctc
    roy.row-i : Error       ➔ nes ◄ row-message-i              for i = 1;n
    roy.row-i = fv          ➔ nes ◄ 'row yoke not satisfied'   for i = 1;n
    let
        new-hea   = tah[co-ide/?]                              new table header
        new-tat   = (new-hea, roy)                             new table type
        new-tab   = (new-ctc, new-tat)                         new table
    'primary' /: com        ➔ (sta, (tre, (dab[ta-ide/new-tab], dbr, mon)) )
    chs = 'restrict'        ➔ nes ◄ 'a parent column can't be removed'
    chs = 'cascade'         ➔ … conditional removal of all children that cause inconsistences
```

Filter the indicated columns of a table (remove the not-indicated)

```
va-filter-col-from-ta : ListOfIde x Identifier ↦ NewInsDen
va-filter-col-from-ta : ListOfIde x Identifier ↦ WfNewSta → WfNewSta
```

In this definition, we refer to the domain of lists of identifiers ListOfIde (Sec. 6.1). We skip its formal definition.

Modify a column in a table conditionally

The constructed denotation modifies all these elements of an indicated column that belong to rows satisfying a given row yoke. The new values in the indicated column are calculated by means of a modifying row yoke. An example of such an instruction may be the following:

```
UPDATE Employees
SET Salary        = Salary * 1,1
WHERE Position    = 'salesman'
```

The denotation of such instruction builds a new table in five steps:

1. it transforms the source column table-content into a row table-content,
2. it calculates the modified values of the target column,
3. it conditionally replaces in all rows the former values in the indicated columns by new values,
4. it checks if the new table content is compatible with the table's (unchanged) type,
5. it checks the consistency of the new database.

```
nid-modify-col-in-ta : Identifier x Identifier x RowYokExpDen x RowYokExpDen ↦ NewInsDen
nid-modify-col-in-ta : Identifier x Identifier x RowYokExpDen x RowYokExpDen ↦
                                                                  WfNewSta ↦ WfNewSta
nid-modify-col-in-ta .(ta-ide, co-ide, mo-ryd, se-ryd) =          mo- modifying, se- selecting
    is-error.nes                        ➔ nes
    let
        (sta, (tre, (dba, dbr, mon))) = nes
    dab.ta-ide = ?                      ➔ nes ◄ 'no such table'
    let
        (ctc, tat)    = dab.ta-ide
        (tah, roy)    = tat
        com           = marking.(tah.co-ide)
```

```
ctc.co-ide = ?                    ➔ nes ◄ 'no such column'
let
   (lar-1,…,lar-k)  = C2R.ctc
   mo-bav-i         = mo-roy.lar-i            for i = 1;k          modified values
   se-bav-i         = se-roy.lar-i                                 selection values
mo-bav-i  : Error                 ➔ nes ◄ mo-bav-i     for i = 1;k
se-bav-i  : Error                 ➔ nes ◄ se-bav-i     for i = 1;k
type.se-bav-i ≠ 'boolean'         ➔ 'boolean value expected'
let
   new-lar-i  =
      se-bav-i = tv               ➔ lar-i[co-ide/mo-bav-i]   for i = 1;k
      true                        ➔ lar-i
   new-rtc          = (new-lar-1,…,new-lar-k)
   new-ctc          = R2C.new-rtc
   tab-message      = type-table-check.(new-ctc, tat)
tab-message : Error               ➔ nes ◄ tab-message
let
   new-tab   = (new-ctc, tat)
   new-col   = new-ctc.co-ide
here we start the consistency check of the database
com = {}                          ➔ (sta, (tre, (dab[ta-ide/new-tab], dbr, mon)))
'primary' : com                   ➔          the modified column is marked to be a parent
   are-repetitions.new-col  ➔ 'repetitions in a parent column'
   let
      orp-message = col-has-orphan.(dab, ta-ide, co-ide)
   orp-message : Error      ➔ nes ◄ orp-message
   orp-message = tv         ➔ nes ◄ 'orphan detected'
   com = {'primary'}        ➔ (sta, (tre, (dab[ta-ide/new-tab], dbr, mon)))   no parents declared
   let
      {pa-ide-1,…,pa-ide-k, 'primary'} = com                               some parents declared
      result-i  = subordination.(dab, ta-ide, pa-ide-i, co-ide)   for i = 1;k
   result-i : Error         ➔ nes ◄ result-i                      for i = 1;k
   true                     ➔ (sta, (tre, (dab[ta-ide/new-tab], dbr, mon)))
'primary' /: com                  ➔
   let
      {pa-ide-1,…,pa-ide-k} = com
      result-i  = subordination.(dab, ta-ide, pa-ide-i, co-ide)   for i = 1;k
   result-i : Error         ➔ nes ◄ result-i                      for i = 1;k
   true                     ➔ (sta, (tre, (dab[ta-ide/new-tab], dbr, mon)))
```

The consistency of a database may be destroyed by our instruction if the marking of the modified column is not empty and:

1. the modified column is marked as a parent and,
   a. it is not repetition free, or
   b. has orphans in the base,
2. the modified column is marked as a child but its table does not satisfy the expected subordination relations.

Note that our instruction may be also used to replace a current value in a table field by a new value. The table field is in such a case identified by a column name together with a row yoke that identifies the row where the field belongs.

11.3.4.3 Transactions

As was announced in Sec. 10.6 *transactions* in SQL are blocs of sequentially composed instructions, i.e., composed instructions, enriched with special instructions protecting databases from the destruction of their integrity. In such cases a programmer in SQL may use a *security instruction* that order the system to make a copy of the current database which the system may bring back if an operation can't be completed without errors.

It seems that in our model the idea of protecting a database against an integrity violation is already built in in our instructions where such violations are signalised by error messages and the state remains unchanged. Nevertheless, for the completeness of our model, we define the instructions of handling security copies.

Create a security copy

```
tra-create-security-copy: Identifier ⟼ NewInsDen
tra-create-security-copy: Identifier ⟼ WfNewSta ⟼ WfNewSta
tra-create-security-copy.ide.nes =
   is-error.nes   ➔ nes
   let
      (sta, (tre, (dba, dbr, mon))) = nes
   dbr.ide = !    ➔ nes ◄ 'identifier already used'
   true           ➔ (sta, (tre, (dba, dbr[ide/dba], mon)))
```

Remove a security copy

```
tra-remove-security-copy : Identifier ⟼ WfNewSta ⟼ WfNewSta
tra-remove-security-copy.ide.nes=
   is-error.nes   ➔ nes
   let
      (sta, (tre, (dba, dbr, mon))) = nes
   dbr.ide = ?    ➔ nes ◄ 'no such security copy'
   true           ➔ (sta, (tre, (dba, dbr[ide/?], mon)))
```

Our last instruction replaces the current database by an indicated security copy and removes the security copy from the repository.

Recover the security copy

```
tra-recover-security-copy : Identifier ⟼ WfNewSta ⟼ WfNewSta
tra-recover-security-copy.ide.nes =
   is-error.nes   ➔ nes
   let
      (sta, (tre, (dba, dbr, mon))) = nes
   dbr.ide = ?    ➔ 'no such security copy'
   let
      sec-dba = dbr.ide
   true           ➔ (sta, (tre, (sec-dba, dbr[ide/?], mon)))
```

11.3.4.4 Queries

A query is an instructions that first creates a new table from another table or tables, and then assigns it to a system identifier monitor in the current database. We skip obvious definitions of their constructors.

11.3.4.5 Instructions modifying integrity constraints

There are basically three categories of integrity-constraint modifications:

1. the modifications of an indicated column yoke of an indicated table,
2. the modification of the row yoke of an indicated table,
3. the modification of the current subordination marks.

The corresponding constructors may be easily defined in our model.

11.3.4.6 Cursors

Cursors (Sec. 10.9) are mechanisms used to get rows from tables. In our model, that can be easily defined, e.g., by adding a column to a table that enumerates its rows.

11.3.4.7 Views

Views are virtually procedures that call table instructions. They may be introduced in our model in a similar way as in **Lingua**.

12 AN EXERCISE WITH A DENOTATIONAL CONCURRENCY

12.1 An overview of our model of concurrency

So far, our programs may be seen as flowchart-like structure where nodes are state-to-state functions. Since we have resigned from **goto**'s, our flowcharts are built by three constructors: sequential composition, branching, and looping. Note that when passing from **Lingua** to **Lingua-SQL**, we only enriched the variety of nodes of our flowcharts, but we preserved flowchart control structures and state-to-state denotations.

To incorporate a concurrency mechanisms into our model we shall go beyond both these paradigms:

- flowcharts will be replaced by Petri nets,
- state-to-state functions, i.e., sets of pairs of states, by sets of finite or infinite sequences of states called *bundles of computations.*

In both these cases we have to do with natural generalizations: Petri nets are generalizations of flowcharts, and sequence — of ordered pairs.

Independently of these modifications we shall also introduce a new *algebra of symbolic behaviors* between the algebra of abstract syntax and that of denotations (Fig. 12.1-1). Before we proceed to building our new model we shall roughly explain the role of the new algebra on the exaple of sequential programs.

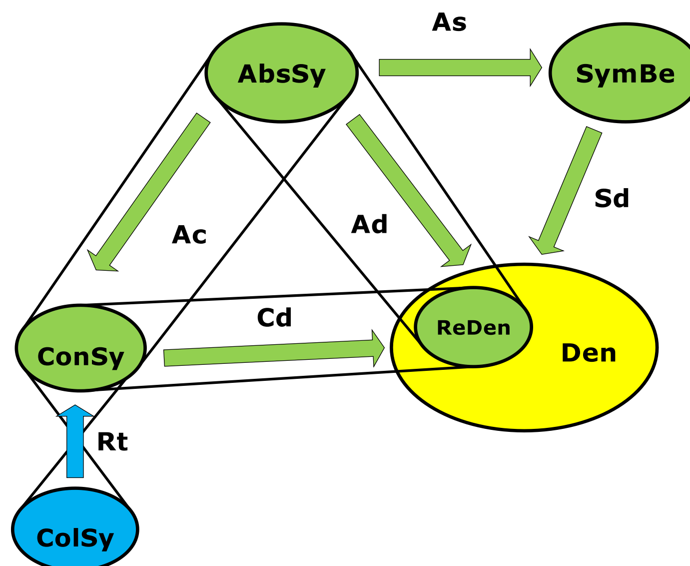


Fig. 12.1-1 An extended algebraic model of a programming language

The elements of the new algebra are called *symbolic behaviors*, and are sets of *symbolic executions*. The latter, in turn, are finite sequences of *atoms*, and atoms are the following program components:

- single declarations of all categories,
- single instructions that include:
  - assignments,
  - procedure calls,
  - value expressions

In which way value expressions are regarded as instructions, will be seen in a moment.

Now, with every program we assign its symbolic behavior, i.e., the set of its potential symbolic executions. From the view point of formal-language theory, behaviors may be regarded as regular languages over an alphabet of atoms. The step from symbolic behaviors to denotations consists in replacing:

- atoms by their denotations,
- sequences of atoms by sequential compositions of these denotations,
- sets of sequences of atoms by set-theoretic unions of the denotations of sequences; here a nondeterminism may come into play.

To describe our new model in a more formal way — still without going into technical details — let us show how it works in the case of instructions. In that case the corresponding domain of symbolic behaviors is defined as follows:

```
sbi  : SymBehIns  = Set.SymExeIns                       symbolic behaviors of instructions
sei  : SymExeIns  = InsAto^{c*}                         symbolic executions of instructions
ati  : AtoIns     = AsgIns | ProCal | ValExp            atoms of instructions
ain  : AsgIns     = assign(AbsRefExp , AbsValExp)       assignment instructions
…
```

Having these domains, we define a *symbolic semantics of instructions* as a function that assigns symbolic behaviors to (abstract-syntax) instructions:

```
SSI : AbsIns ⟼ SymBehIns

SSI[ati]                          = {(ati)}
SSI[sin1 ; sin2]                  = SSI[sin1] © SSI[sin2]
SSI[if exp then ins1 else ins2 fi] = ({(exp)} © SSI[ins1]) | ({(not exp)} © SSI[ins2])
SSI[while exp do ins od]          = ({(exp)} © SSI[ins1])^{c*} © {(not exp)}
```

where © is concatenation of languages (Sec. 2.5).

Once we are done with the mechanism of symbolic behaviors, we can pass to the level of denotations. We define a function from symbolic behaviors to state-to-state denotations:

```
S2D : SymBehIns ⟼ InsDen
```

First, with every (atomic) behavior that includes only one execution with only one atom we assign the (earlier defined) state-to-state denotations of this atom. E.g., (cf. Sec. 7.1)

```
S2D[{(assign(AbsRefExp , AbsValExp))}] = A2D[assign(AbsRefExp , AbsValExp)]
```

Next, with every (abstract) value expression ave we assign the following state-to-state function that we shall call a *filter*:

```
S2D[{(ave)}] : State → State
S2D[{(ave)}].sta =
   A2D[ave].sta = ?    ➔ ?
   let
      val = A2D[ave].sta
   val : Error          ➔ sta ◄ val
   val /: Bool          ➔ sta ◄ 'boolean expected'
   val = tv             ➔ sta
   val = fv             ➔ ?
```

The denotation of {(ave)} is a subset of the identity function on states. It is transparent for states evaluating ave to tv and otherwise generates an error or is undefined accordingly. The role of subsets of identity in the semantics of programs is explained in Sec. 8.3.

Finally, with every symbolic execution of an instruction we assign the (functional) composition of all the denotations of its elements, and with every symbolic behavior — the (set theoretic) union of the denotations of its executions:

S2D.{(ati-1,…,ati-n) | (ati-i,…,ati-n) : sbi} = for all sbi : SymBehIns

U {(S2D[ati-1] ● … ● S2D[ati-n] | (ati-i,…,ati-n) : sbi}

On the ground of our extended model of sequential languages the step from sequentiality to concurrency consists in:

- we replace symbolic behaviors, i.e., regular languages over an alphabet of atoms generated by flowcharts by *trace languages* of Antony Mazurkiewicz (see [73]) generated by simple Petri nets,
- we replace state-to-state denotations by bundles of computations of Andrzej Blikle (see [21], [22] and [24]).

Similarly as in the case of SQL, we shall limit our investigations to the denotational algebra of the new model, and we shall only sketch the construction rather than go into "practical technicalities".

12.2 Bundles of computations

12.2.1 Abstract nets and quasinets

In Sec. 2.5 and Sec. 2.7 we have described a CPO of formal languages and of binary relations respectively. Both provide an adequate context for equational (fixed-point) descriptions of the syntaxes and the denotations of programming languages. Both are equipped with a continuous monoid operation: the concatenation of languages and the sequential composition of relations respectively. In our approach to concurrency we shall use yet another CPO with a monoid operation: a CPO of bundles of computations. All three are particular cases of abstract *nets* and *quasinets* investigated by A.Blikle in the decade of 1970. in several papers (cf. [19], [21], [22], [23] and [24]). In this section we give a short introduction to their theory.

A *quasinet* is a partially ordered sets with a monoidal operation. By a *monoid* over a set A we mean a triple

(A, ●, e)

where ● is a total function, called *composition*

● : A x A ⟼ A

with two following properties

1. (a ● b) ● c = a ● (b ● c) — *associativity*
2. a ● e = e ● a = a — e is the *unit* of composition

By a *quasinet* we mean a quintuple (A, ⊑, ●, Φ, e) with the following properties[107]:

(1) (A, ⊑, Φ) is a CPO
(2) (A, ●, e) is a monoid
(3) ● is continuous
(4) Φ ● a = Φ for any a : A

A quasinet is said to be a *net* if

(5) a ● Φ = Φ for any a : A.

A quasinet is said to be *set-theoretic* if A is a set of sets and ⊑ is an inclusion of sets. If there exist elements a and b both different from Φ such that

a ● b = Φ

---

[107] Nets were originally introduced by A.Blikle in [19] as complete lattices with a monoidal operation. Here we are weakening our definition assuming that a net must be a CPO, since in fact all what we need is that continuous fixed-point equations have least solutions. Besides, set-theoretic CPO's of functions are not lattices since a union of two functions needs not to be a function.

then they are called the *divisors of zero*.

Originally quasinets were defined as complete lattices, rather than CPO's. A *complete lattice* is a POS (A, ⊑) where for every subset of A there exists the least upper bound and the greatest lower bound. Here we introduce nets and quasinets understood as POS's in order to cover POS's of partial functions which do not constitute lattices.

12.2.2 Nets of the bundles of computations

In the majority of denotational models of programming languages, the denotations of programs are state-to-state functions, or — in non-deterministic cases — binary relations on states. This approach is adequate to situations where programs are supposed to terminate after a finite number of staps. If, however, we intend to deal with programs that "run forever" and we still regard them correct — as it is frequently the case with concurrent programs — binary relations become inadequate. In the first place on their ground we can't describe a situation where a nondeterministic program starting with a given input state may generate a terminating execution or alternatively an infinite execution. Besides, on the ground of relational semantics we can hardly talk about temporal properties of executions.

To tackle the mentioned problems we shall use a model where the denotations of programs are sets of sequences of states generated by programs. Such models were investigated by A. Blikle in [21], [22] and [24].

Let State be an arbitrary set of items called *states*. By a *computation* over State we mean any empty, finite, or infinite sequence of states. Let

com : Co.State = FiCo.State | InCo.State — the set of all computations over State
com : FiCo.State = $State^{c*}$ — the set of all finite computations over State
com : InCo.State = $State^{c\infty}$ — the set of all infinite computations over State including the empty computation ()

Consider two computations (sta-11,…,sta-1n) and (sta-21,…,sta-2m) with n,m ≤ ∞, each of which may be empty (n=0 or m=0), finite (1≤n≤k or 1≤m≤k) or infinite (n=∞ or m=∞). By a *sequential composition* or just a *composition* of these computations we mean a computation defined in the following way:

```
(sta-11,…,sta-1n) ● (sta-21,…,sta-2m) =
   n = ∞            ➔ (sta-11,…,sta-1n)
   n = 0            ➔ ()
   m = 0            ➔ ()
   sta-1n ≠ sta-21  ➔ ()
   true             ➔ (sta-11,…,sta-1n, sta-22,…,sta-2m)
```

Intuitively a sequential composition of two computations is a computation that starts with the first one, and continuous with the second. In turn, an empty computation () is a computation that "cannot happen". With this interpretation an intuitive explanation of our definition is the following:

- an infinite computation followed by any other computation (even empty), is the former computation, because nothing may be added "at the end" of an infinite computation,
- a composition of a computation that can't happen with any computation is a computation that can't happen,
- a sequential composition of a finite computation with a computation that can't happen, cannot happen,
- a sequential composition of a finite computation whose last state is different from the first state of the second computation, i.e., where sta-1n ≠ sta-21, can't happen because the second is supposed to be a continuation of the former,

- finally, if sta-1n = sta-21, then the second computation continues the first computation.

As is easy to check, our composition is associative, and the following equations are satisfied:

() ● com = () for any com : Co.State
com ● () = () for any com : FiCo.State
com ● () = com for any com : InCo.State

By a *bundle of computations* we mean any set of computations which includes the empty computation. We shall denote:

Bun.State = { P | P ⊆ Co.State **and** () : P}[108]
-A = State – A — for any set of states A ⊆ State
[A] = {(sta) | sta : A} | {()} — for any set of states A ⊆ State; [A] is called a *test*
{P} — the set of all states that appear in bundle P, e.g. {[A]} = A
Ib = [State] — *identity bundle*
Eb = {()} — *empty bundle*
fin.P = P ∩ State^{c*} — *finitistic part of* P
inf.P = (P – fin.P) | {()} — *infinitistic part of* P

In our new model bundles will represent denotations of programs. A bundle is said to be *infinitistic* if it includes infinite computations. The *composition of bundles* is defined as follows:

P ● Q = {com-1 ● com-2 | com-1 : P **and** com-2 : Q}

As is easy to check:

(Bun.State, ⊆, ●, Eb, Ib)

is a set-theoretic quasinet[109], but not a net since

P ● Eb = inf.P

Similarly as in the case of languages, quasinets of bundles constitute complete lattices.

In obvious contexts we shall allow to omit ● and write com-1com-2 and PQ respectively. We assume also that composition binds stronger than union, i.e.

PQ | RF = (PQ) | (RF)

Empty bundle Eb represents a program that "can't run". A union of bundles P|Q represents a (possibly) nondeterministic branching of programs P and Q, and a composition PQ represents a sequential composition of these programs. A computation consisting of two states only (sta-1, sta-2) is called an *atomic computation*. A bundle whose all non-empty computations are atomic is called an *atomic bundle*.

The powers of a bundles — P^n, P^+ and P^* — are defined according to the rule indicated for abstract quasinets (Sec. 12.2.1). As is easy to see:

$P^+ = ∪\{ P^n \mid n = 1,2,…\}$.

It should be observed that in the quasinet of bundles we have so called *zero divisors*, i.e., such P and Q both different from Eb, that

PQ = Eb.

E.g., if A ∩ B = {}, then [A][B] = Eb. As is easy to check (see [21] and [24]) the following equations are true for any bundles P, Q and R:

[A]P = {com | com : P **and** first.com : A} | {()}

---

108 Bundles will be denoted by P, Q, R,… and sets of states by A, B, C,…

109 Note that if we had not assumed that all bundles include empty computation, then the composition of an infinitistic bundle with the empty bundle would be the empty bundle, which would mean that if we compose sequentially a program which may loop indefinitely with a program that can't run, then the composed program can't run.

P(Q|R) = PQ | PR — *left distributivity over union*
(Q|R)P = QP | RP — *right distributivity over union*

In the quasinet of bundles we can define constructors that correspond to structured constructors of programs:

P ; Q = PQ
if (T, F) **then** P **else** Q **fi** = TP | FQ

In the second equation we assume that $T, F \subseteq [State]$ are disjoint subsets of identity bundle that represent a three-valued predicate. Note that if we consider a bundle of the form

R **; if** (T, F) **then** P **else** Q **fi** = R (TP | FQ)

then a computation of R which terminates with a state in T composed with a computation of FQ results an empty computations, because it "cannot happen".

To deal with infinite computations we introduce a composition of an infinite sequence of bundles, informally:

$P_1 \bullet P_2 \bullet \dots$

To define this operation we need a few auxiliary concepts. We say that com-1 is a *prefix* of com-2, in symbols

com-1 ⊑ com-2

if there exists com, such that com-2 = com-1 ● com. Note that

() ⊑ ()

but

() ⊑ com does not hold for com ≠ (). (12.2.2-1)

A composition of an infinite sequence of computations denoted by

C.(com-1, com-2, …)

is the shortest[110] com such that (∀ n) (com-1 ● … ● com-n ⊑ com). Note that if for some n ≥ 1,

com-1 ● … ● com-n = (),

then

C.(com-1, com-2, …) = ().

As we see, a composition of an infinite sequence of computations may be empty, finite or infinite. Now, we can define a composition of an infinite sequence of bundles:

C.(P_1, P_2, …) = { C.(com-1, com-2, …) | (∀ i) com-i : P_i }

and an infinite power of a bundle

P^∞ = **C**.(P, P, …)

Using the introduced notation we can define the denotation of a while-loop:

while (T, F) **do** P **od** = (TP)*F | $(TP)^\infty$

In this case (TP)*F is the least solution of the equation

X = TPX | F

which can be easily proved using Kleene's theorem 12.2.1-1. At the same time, if all nonempty computations in $(TP)^\infty$ are infinite, then $(TP)^\infty$ is the greatest solution of the equation

[110] Note that if starting with some n all com_i are tests then C.(com_1, com_2, …) is finite and is either empty or equal to $com_1 \bullet \dots \bullet com_{n-1}$.

X = TPX

Proof in [24].

12.2.3 Strong total correctness of bundles

Bundles constitute an adequate framework for the development of a Hoare-like correctness theory for strong total correctness with clean termination, i.e. without abortion (cf. Sec. 8.7). In **Lingua**, abortion states carry an error message in their error registers and in this way indicate that a program has stopped its execution and displays an error message. In the current abstract model of states we shall only assume the existence of a distinguished subset of the set of states whose elements are called *abortion states.*

Abort ⊆ State

We will say that a program terminates its execution *cleanly* if its terminal state is not an abortion state.

Now, let A be an arbitrary subset of State, and let P be an arbitrary bundle over State. We define two operations of *left-* and respectively *right composition* of a bundle with a set of states.

A ● P = { sta-n | (∃ (sta-1,…,sta-n) : P) sta-1 : A }
P ● B = { sta-1 | (∃ (sta-1,…,sta-n) : P) sta-n : B }

Here we use the same symbol as for the composition of bundles and we allow omitting this symbol thus writing AP and PB. As is easy to prove both these operations are monotone and associative.

Lemma (12.2.3-1) *For any* A, B ⊆ State *and any* P, Q : Bun.State

if A ⊆ B *then* AP ⊆ BP *and* PA ⊆ PB
if P ⊆ Q *then* AP ⊆ AQ *and* PA ⊆ QA
A(PQ) = (AP)Q and (PQ)A = P(QA) ■

Using our operations we can define the properties of partial correctness and of weak total correctness of bundles:

AP ⊆ B — *partial correctness* of P for precondition A and postconditions B; every finite computation of P that starts in A, terminates in B, but there may be infinite computations that start in A.

This property corresponds to a partial correctness in the sense of C.A.R. Hoare (cf. [5] and [61]).

A ⊆ PB — *weak total correctness* of P for precondition A and postconditions B, for every a : A there exists a computation in P that starts with a, and terminates in B.

This property is called <u>weak</u> total correctness, since it only guarantees that for any s in A <u>there exists</u> a finite computation that starts with s and terminates in B, but there may be other computations that start with s, but either terminate outside B or do not terminate at all[111]. Of course, in the case of deterministic programs weak total correctness is just total correctness, and if we assume that B does not include states that carry an error message, then it is a *clean total correctness*. In the deterministic case clean total correctness implies partial correctness.

In the case of nondeterministic programs with possibly infinite computations we need a stronger concept of correctness which would guarantee that all computations that start in A terminate in B. To define this concept we introduce a *strong composition* of a bundle with a set of states:

P ■ B = {sta | (∀com : P) if first.com = sta than com : State^{c*} and last.com : B}

where first.com and last.com are the first and the last element of com.

P ■ B is the set of all states which give rise to finite computations only, and all these computations terminate in B. Consequently, if sta : P ■ B then all computations that start with sta are finite and terminate in B. Similarly to the properties (12.2.3-1) also now we have monotonicity and associativity:

---

[111] The idea to call this total correctness a weak total correctness is due to Krzysztof Apt (personal communication).

Lemma (12.2.3-2) *For any* A, B ⊆ State *and any* P, Q : Bun.State

if A ⊆ B *then* P ■ A ⊆ P ■ B
if P ⊆ Q *then* P ■ A ⊆ Q ■ A
(PQ) ■ A = P ■ (Q ■ A) ■

Now, the strong correctness is defined as follows:

A ⊆ P ■ B — *strong correctness* of P for precondition A and postcondition B; all computations of P that start in A terminate in B.

If B includes no error states then we say that P is strongly correct with *clean termination*. Of course, in a deterministic case weak total correctness is equivalent to strong correctness, but in a nondeterministic case the former is weaker than the latter. The following obvious lemma is useful in developing proof rules for structured constructors:

Lemma 12.2.3-3 *For any* A, B ⊆ State *and* P : Bun.State

A ⊆ P ■ B *iff* AP ⊆ B *and* A ⊆ P ■ State ■

Having defined strong total correctness of bundles we can formulate Hoare-like proof rules similar as in the case of binary relations (cf. Sec. 8.7).

Lemma 12.2.3-4 *For any* A,D ⊆ State *and* P,Q : Bun.State

there exist B, C ⊆ State *such that*
(1) A ⊆ P ■ B
(2) B ⊆ C
(3) C ⊆ Q ■ D

---

(4) A ⊆ (PQ) ■ D

Proof If (1) – (3) are satisfied then partial correctness A(PQ) ⊆ D is immediate from Lemmas 12.2.3-1 and 12.2.3-3. The termination is obvious. In turn, if (4) is satisfied then by Lemma 12.2.3-2, A ⊆ P ■ (Q ■ D) and setting B = C = Q ■ D we get the proof.

■

Lemma 12.2.3-5 *For any predicate* (T, F) any A,B ⊆ State *and* P,Q : Bun.State

(1) A ∩ {T} ⊆ P ■ B
(2) A ∩ {F} ⊆ Q ■ B
(3) A ⊆ {T} | {F}

---

(4) A ⊆ **if** (T, F) **then** P **else** Q **fi** ■ B

Since

if (T, F) **then** P **else** Q **fi** = TP | FQ,

the proof is analogous to the former. Of course, in the case of classical predicates T | F = State, and therefore condition (3) is a tautology and may be omitted.

We skip the discussion of a while-loop to avoid technicalities that would go beyond the scope of this section. Such rules for the case of I-O functions have been discussed in [28] and in Sec. 8.7.2.

Two important issues have to pointed out at the end. First concerns the fact that we are not building here any formalized logic of programs like in [61]or [4]. Our proof rules are just lemmas proved on the ground of set theory to be used in proving properties of programs on the same ground. Logic involved in these proofs is just a usual mathematical logic.

The second issue concerns the fact that our proof rules may be regarded as construction rules of correct programs analogously as in Sec. 9.4.

12.2.4 Temporal quantifiers

Temporal quantifiers may be easily defined in the model of bundles. For that sake let us regard computations as functions that map nonnegative integers into states. We assume further that each such function is defined on an initial interval of the form [1,…,n]. By dom.com, where com is a computation, we shall denote the domain of com. Let A,B ⊆ State represent the truth parts of a predicate, let com be an arbitrary computation, and let i and j run over dom.com. Then:

A □ com	iff	(∀ i) com.i : A	— *always* A
A ◊ com	iff	(∃ i) com.i : A	— *eventually* A
(A 𝒰 B) com	iff	(∃ j) ((∀ i ≤ j) com.i : A) **and** com.j : B	— A *until* B
(A 𝒲 B) com	iff	(A 𝒰 B) com **or** (**not** (B ◊ com) **and** (A □ com))	— A *unless* B
(A → B) com	iff	**if** (∃ i) com.i : A **then** (∃ j ≥ i) com.j : B	— *if* A *then later* B
(A ← B) com	iff	**if** (∃ i) com.i : B **then** (∃ j ≤ i) com.j : A	— B *only if earlier* A

The quantifier 𝒲 is called by Mordechai Ben-Ari in [13] a *weak until* because it does not require that B eventually becomes true. In such a case A remains true forever. Our quantifiers may be easily generalized to bundles:

A □ P iff (∀ com : P) A □ com

and analogously for other quantifiers. We say that A is *hereditary* in a bundle P, in symbols

A ► P

if

(∀ com : P) either -A □ com or (∃ i) (∀ j ≥i) (com.j : A)

12.3 Petri nets and trace languages

12.3.1 Trace languages of Antoni Mazurkiewicz

Let Alp be a finite or infinite alphabet. By a *dependency relation* or simply a *dependency* in Alp we mean any finite binary relation D ⊆ Alp x Alp such that, if (a, b) : D then (b, a) and (a, a) : D. With every dependency we associate its alphabet alp.D which is the set of all letters that appear in D. Consequently, a dependency D is reflexive and symmetric in alp.D. A dependency D is said to be a *full dependency* if

$D = (alp.D)^{c2}$

By Dep.Alp we shall denote the set of all dependencies over Alp. Dependencies have the following important properties:

1. empty relations, identity relations and full relations in their alphabets are dependencies,
2. union and intersection of a finite number of dependencies is a dependency,
3. every dependency is a finite union of full dependencies.

For instance,

$D = \{a, b\}^{c2} \mid \{a, c\}^{c2}$

is a dependency relation over alp.D = {a, b, c}. By an *independency relation induced by* D we mean a relation

$ind.D = (alp.D)^{c2} - D.$

Clearly every independency is symmetric and irreflexive in its alphabet alp.(ind.D). In the case of our example ind.D = {(b, c), (c, b)}, and alp.(ind.D) = {b, c}. Independency relations may be described as a symmetric closures, abbreviated sc, of a non-symmetric relations. E.g.,

ind.D = sc.{(b, c)}.

The alphabet alp.(ind.D) may be a proper subset of alp.D, but may also be equal to alp.D. E.g. if D is an identity relation, and alp.D includes at least two elements, then alp.D = alp.(ind.D).

Given a dependency D we define an equivalence relation $\equiv_D$ between words over alp.D as the least congruence in the monoid of these words such that for any a, b : alp.D:

if (a, b) : ind.D then ab $\equiv_D$ ba (12.3.1-1)

This relation is called *trace equivalence* for D. Equivalence classes over $\equiv_D$ in (alp.D)* are called *traces* over D, and constitute a quotient monoid (alp.D)*/$\equiv_D$. An element of that monoid represented by a word w will be denoted by $[w]_D$ or simply by [w] if D is understood. In that case w is said to be a *representant* of [w]. In the general case a trace may have many representants, but

Fact 12.3.1-1 If a dependency D is full, then each D-trace has exactly one representant.

By

Tra.D = {[w] | w : $(alp.D)^{c*}$}

we denote the set of all traces over D. If by ● we denote both the concatenation of words and the concatenation of traces then the latter is defined as follows:

$[w_1]_D$ ● $[w_2]_D$ = $[w_1$ ● $w_2]_D$

where w_1, w_2 are words over alp.D. By the definition of traces, [w] is the set of all words that arise from w by the permutations of all adjacent independent letters. For instance, in the case of our example

[abbca] = {abbca, abcba, acbba}.

For any dependency D, by a *trace language* over D we mean any set of traces over D. From now on, "usual" languages described in Sec. 2.5 will be called *word languages* or just *languages*. For any word language L over alp.D we define the corresponding trace language

$[L]_D$ = {$[w]_D$ | w : L}

In this case L is called a *representant* of $[L]_D$. Of course, if D is not a full relation then $[L]_D$ has more than one representant.

If D is understood, then we simply write [L]. By TraLan.Alp we shall denote the set of all trace languages over Alp, i.e.,

TraLan.Alp = {$[L]_D$ | L : Lan.Alp, D : Dep.Alp}

Now, with every trace language TL we can assign a word language which includes all words belonging to the traces of TL. A function which transforms trace languages into word languages is the following

T2L : TraLan.Alp ⟼ Lan.Alp
T2L.TL = U {t | t : TL}

Since traces are classes of abstraction, hence sets, their unions makes sense. Note that

L ⊆ T2L.$[L]_D$ (12.3.1-2)

but T2L.$[L]_D$ may be significantly larger than L because it includes all words created by the permutations of adjacent independent letters in the words of L. Language T2L.$[L]_D$ is called the D-*completion* of L.

Fact 12.3.1-2 For every L and D, T2L.$[L]_D$ is the largest representant of $[L]_D$.

Fact 12.3.1-3 For every L and D, T2L.$[T2L.[L]_D]_D$ = T2L.$[L]_D$

If

$L = T2L.[L]_D$

then L is said to be D-*complete*. Consider the following example :

L = {abbca, abc}
D = $\{a, b\}^{c2}$ | $\{a, c\}^{c2}$ hence Ind = {(b, c), (c, b)}

In this case

[L] = { [abbca], [abc] }
[abbca] = {abbca, abcba, acbba}
[abc] = {abc, acb}
T2L.[L] = {abbca, abcba, acbba, abc, acb}

Fact 12.3.1-4 If D is full than any word language over alp.D is D-complete

The last fact implies that trace languages are natural generalizations of word languages. If D is full then $[L]_D$ is said to be D-*sequential*. Of course, each D-sequential language is D-complete but not vice versa.

For every dependency relation D, the set of all trace languages over D, i.e.,

TraLan.D = $\{[L]_D$ | L : Lan.(alp.D)}

constitutes a monoid with the operation

$[L_1] \bullet [L_2] = \{[w_1] \bullet [w_2] \mid w_1 : L_1, w_2 : L_2\}$

and unit [()] = {()}. We shall allow writing $[L_1][L_2]$ for $[L_1]\bullet[L_2]$ and analogously for traces. We also set:

$[L]^0 = [()]$
$[L]^{n+1} = L^n \bullet [L]$ for n = 0,1,…
$[L]^* = U \{[L]^n \mid n = 0,1,…\}$

Theorem 12.3.1-1 (see [72]) For any dependency relation D, any word languages L_1 and L_2 over alp.D, and any family of word languages $\{L_i \mid i = 1,2,…\}$ over alp.D the following properties hold:

(1) [{}] = {}
(2) $[L_1][L_2] = [L_1L_2]$
(3) $[L_1] \mid [L_2] = [L_1|L_2]$
(4) $U \{[L_i] \mid i= 1,2,…\} = [U \{L_i \mid i = 1,2,…\}]$
(5) $[L]^* = [L^*]$
(6) if $L_1 \subseteq L_2$ then $[L_1] \subseteq [L_2]$ ■

Note that in (6) a converse implication does not need to be true. E.g.

[{abbca, abcba}] ⊆ [{abbca}]

It holds, however, if L_1 and L_2 are the largest representants of respective trace languages. Indeed, let w : L_1. In that case [w] : $[L_2]$. Let [w] = $\{w_1,…,w_n\}$. Since L_2 is the largest representant of $[L_2]$, all w_i's must belong to L_2, and, therefore, w : L_2.

Theorem 12.3.1-2 For any dependency relation D the set TraLan.D of all trace languages over this relation is a set-theoretic net with concatenation of trace languages as the monoid operation, with the unit of the monoid [()] and the least element {}. ■

Proof First note the following facts:

- the fact that (TraLan.D, ●, [()]) is a monoid is obvious from (2) of Theorem 12.3.1-1,
- the fact that (TraLan.D, ⊆) is a CPO, follows from (4),
- the equalities [L][()] = [()][L] follow from (1) and (2).

To prove that ● is continuous — i.e. that it is continuous in each of its arguments separately — consider a chain of trace languages:

$[L_1] \subseteq [L_2] \subseteq …$

where all L_i's are the largest representants of the respective trace languages. Let Q be the largest representant of [Q]. To prove that ● is continuous in second argument we have to prove that

[Q] ● U {[L_i] | i = 1,2,…} = U {[Q] ● [L_i] | i = 1,2,…}

Indeed, by (2) and (4) and the continuity of the concatenations of word languages, we have

[Q] ● U {[L_i] | i = 1,2,…} = [Q] ● [U {L_i | i = 1,2,…}] = [Q ● U {L_i | i = 1,2,…}] =

[U { Q ● L_i | i = 1,2,…}] = U { [Q ● L_i] | i = 1,2,…}] = U {[Q] ● [L_i] | i = 1,2,…}.

The proof for the first argument is analogous. ■

Now, we can proceed to the definitions of synchronization operations of word languages and trace languages. We start from the notion of projection. Let B ⊆ Alp, and let w be a word over Alp. By a *projection* of a word w over the alphabet B we mean a word over B defined as follows:

```
pro.(B, Alp) : Word.Alp ⟼ Word.B
pro.(B, Alp).w =
   w = ()    ➔ ()
   let
      z ● a = w       where a : Alp
   a /: B    ➔ pro.(B, Alp).z
   a  : B    ➔ (pro.(B, Alp)) ● a
```

Projection function removes from w all letters which are not in B. This function may be extended to languages in an obvious way:

pro.(B, Alp).L = {pro.(B, Alp).w | w : L}

Let now C and D be dependencies and let C ⊆ D. By a *trace projection* of a trace t on dependency C we mean a trace over C defined as follows:

```
tr-pro.(C, D) : Tra.D ⟼ Tra.C

tr-pro.(C, D).t =
   t = [()]D       ➔ [()]C
   let
      p ● [a]D = t    where    a : alp.D
   a /: alp.C   ➔ tr-pro.(C, D).p
   a  : alp.C   ➔ (tr-pro.(C, D).p) ● [a]C
```

This function satisfies the following equality for any w : alp.D:

tr-pro.(C, D).$[w]_D$ = $[\text{pro.(alp.C, alp.D).w}]_C$

As we see, trace projection given a trace over D removes from its representants all letter which are not in alp.C and restricts the dependency to C. Here two cases are possible.

If alp.C is strictly included in alp.D, then traces over C, regarded as sets, are not comparable with traces over D.

Example 12.3.1-1

Let

D = $\{a, b\}^{c2}$ | $\{a, c\}^{c2}$ | $\{a, d\}^{c2}$ and then ind.D = { (b, c), (c, b), (b, d), (d, b), (c, d), (d, c) }
C = $\{a, b\}^{c2}$ | $\{a, c\}^{c2}$ and then ind.C = { (b, c), (c, b) }

In this case ind.C ⊆ ind.D and, e.g.

$[abcd]_D$ = {abcd, abdc, acbd, acdb, adcb, adbc}.
tr-pro.(C, D).$[abcd]_D$ = $[abc]_C$ = {abc, acb}

If, however, alp.C = alp.D then ind.C may be larger than ind.D since C gives "more freedom" for the permutation of letters. E.g.:

$D = \{a, b\}^{c2} \mid \{a, c\}^{c2} \mid \{c, b\}^{c2} = \{a, b, c\}^{c2}$ and then ind.D = { }
$C = \{a, b\}^{c2} \mid \{a, c\}^{c2}$ and then ind.C = {(b, c), (c, b)}.

In this case

$[abc]_D = \{abc\}$
$tr\text{-}pro.(C, D).[abc]_D = [abc]_C = \{abc, acb\}$

and

$[abc]_D \subseteq tr\text{-}pro.(C, D).[abc]_D$

Example 12.3.1-2

Let

$D = \{a, b\}^{c2} \mid \{a, c\}^{c2}$, alp.D = {a, b, c}, ind.D = {(b, c)}
$C = \{b\}^{c2} \mid \{c\}^{c2}$, alp.C = {a, c}, ind.C = {(b, c)}

In this case

$[bac]_D = \{bac\}$
$tr\text{-}pro.(C,D).[bac]_D = [bc]_D = \{bc, cb\}$

Poprosiłem Andrzeja o uzupełnienie. ???

Consider now two word languages L_1 and L_2. By a *synchronization* of these languages we mean a word language over $Alp = alp.L_1 \mid alp.L_2$ defined as follows:

$L_1 \parallel L_2 = \{w \mid w : Alp^*, pro.(alp.L_i, Alp).w : L_i, \ i = 1,2\}$

Theorem 12.3.1-3 (see [72]) The synchronization of word languages is commutative, associative and distributive over arbitrary unions, i.e. for any word languages L, L_1, L_2, L_3, and any family of word languages $\{L_i \mid i : Ind\}$ over a common alphabet:

$L_1 \parallel L_2 = L_2 \parallel L_1$

$L_1 \parallel (L_2 \parallel L_3) = (L_1 \parallel L_2) \parallel L_3$

$(U \{L_i \mid i : Ind\}) \parallel L = U \{L_i \parallel L \mid i : Ind\}$ ■

Consider two trace languages TL_1 and TL_2 over two dependencies D_1 and D_2 respectively. By a *synchronization* of these trace languages we mean a trace language over alphabet $Alp = alp.D_1 \mid alp.D_2$ and dependency $D = D_1 \mid D_2$ defined as follows

$TL_1 \parallel TL_2 = \{[w] \mid w : Alp^*, tr\text{-}pro.(D_i, D).t : TL_i, i = 1,2\}$

Theorem 12.3.1-4 (see [72]) For any dependencies D_1 and D_2, and any word languages L_1 and L_2 over $alp.D_1$ and $alp.D_2$ respectively the following equality holds:

$[L_1]_{D_1} \parallel [L_2]_{D_2} = [L_1 \parallel L_2]_{D_1 \mid D_2}$ ■

Theorem 12.3.1-5 (see [72]) The synchronization of trace languages is commutative, associative and distributive over arbitrary unions, i.e. for any trace languages TL, TL_1, TL_2, TL_3, and any family of trace languages $\{TL_i \mid i : Ind\}$ over a common alphabet:

$TL_1 \parallel TL_2 = TL_2 \parallel TL_1$

$TL_1 \parallel (TL_2 \parallel TL_3) = (TL_1 \parallel TL_2) \parallel TL_3$

$(U \{TL_i \mid i : Ind\}) \parallel TL = U \{TL_i \parallel TL \mid i : Ind\}$ ■

12.3.2 Trace languages and Petri Nets

One of the most important fields of applications of trace languages are Petri nets. Trace languages play the same role for Petri nets, as regular word languages for iterative programs (programs without recursive procedures). Below we sketch this construction following again [72]. By a *Petri net* we mean a quadruple

net = (Place, Transition, Flow, Ini)

where

pla : Place	a finite set of elements called *places*
tra : Transition	a finite set of elements called *transitions*
Flow ⊆ Place x Transition | Transition x Place	*flow relation*
Ini ⊆ Place	*initial marking*

and where the following assumptions are satisfied

Place ∩ Transition = {}	
Flow ∩ $Flow^{-1}$ = {}	no place is an entry and an exit of the same transition
dom.Flow | cod.Flow = Pla | Tra	there are no isolated places or transitions

Now, let for any tra : Transition

Entry.tra = {pla | (pla, tra) : Flow}	*entries* of transition tra
Exit.tra = {pla | (tra, pla) : Flow}	*exits* of transition tra
Neigh.tra = Entry.tra | Exit.tra	*neighborhood* of transition tra

By a *marking* of a net we mean any set of places of this net including an empty set:

mar : Marking = Sub.Place

With every transition we assign a partial *transition function* of the net which describes the transformation of an input marking into an output marking that occurs when this transition is executed (fired).

Tf : Transition ↦ Marking → Marking

A. Mazurkiewicz defines this function in the following way:

$Tf.tra.mar_{in} = mar_{fi}$ iff (in – initial, fi – final)
 Entry.tra ⊆ mar_{in} **and** all entries of tra are marked
 Exit.tra ∩ mar_{in} = {} **and** no exit of tra is marked
 mar_{fi} = Exit.tra | (mar_{in} – Entry.tra)

An explicit definition of this function may be the following

Tf.tra.mar =
 Entry.tra ⊆ mar **and** Exit.tra ∩ mar = {} ➔ Exit.tra | (mar – Entry.tra)
 true ➔ ?

As we see, if all entry places of tra are marked, and no exit places of tra are marked, then the output marking exists and consists of all exit places of tra plus the "unused" places of the initial marking, i.e., mar – Entry.tra. In that case we say that marking mar *enables transition* tra.

Two particular cases of this definition are to be pointed out.

If Entry.tra = {} — in this case we say that tra is an *orphan* — then tra is enabled by any marking mar disjoint with Exit.tra including the empty marking. In this case

Tf.tra.mar = Exit.tra | mar
Tf.tra.{} = Exit.tra

The interpretation of these facts are quite natural. Entry and exit places of a transit define a necessary and sufficient condition to fire that transit. This condition allows for a firing of a transition if all entry places are

marked and no exit places are marked. No entry places and empty marking (hence no marks in exit places) means that the firing condition is satisfied.

The second case to be considered is when Exit.tra = {}. In that case tra is said to be a *widow*, and for any marking mar which satisfies Entry.tra ⊆ mar we have

Tf.tra.mar = mar – Entry.tra

In particular

Tf.tra.(Entry.tra) = {}

By a *path of transitions* or simply a *path,* we mean any finite, possibly empty, sequence of transitions:

pat : Path = Transition^{c*}

Now, we can generalize the single-step function Tf to a many-steps function called the *reachability function* and defined as follows:

Rf : Path ⟼ Marking → Marking

Rf.(tra_1,…,tra_n).mar =
  n = 0 ➔ mar
  true ➔ Tf.tra_n.(Rf.(tra_1,…,tra_{n-1}).mar)

Notice that since Tr.tra is a partial function on markings, so is Rf.pat. If Rf.pat.mar is defined then pat is said to be a *symbolic execution* from mar to Rf.pat.mar. If mar = Ini, then pat is called an *initial execution*. By the *sequential symbolic behavior* of a Petri net, in symbols Sbe.net we mean the set of all initial executions of net:

Sbe.net = {pat | Rf.pat.Ini = !}

Notice that Sbe.net always includes empty path and is prefix closed. The latter property allows for a representation of infinitistic behaviors of nets by infinite branches (see Sec. 12.2.2) of initial executions

$pat_1 \sqsubset pat_2 \sqsubset \dots$

Two cases are of particular interest for further investigations. They correspond to *atomic nets*. A net is said to be *atomic* if it includes one place only. Atomic nets may be marked or unmarked. If such a net is marked then the initial marking of that net consists of its unique place. Otherwise we assume that the initial marking is empty. Now, consider two such nets on Fig. 12.3-1 where A and B represent any finite or empty sets of transitions.

Fig. 12.3-1 Two atomic nets

The sequential behaviors of these nets are the following languages:

(BA)*(B | {()}) — first net
(AB)*(A | {()}) — second net

If A and B are not empty, both these nets represent loops which is due to the fact that the marking generated by widow B is empty, and, in turn, empty marking fires orphan A. Due to the fact that both languages are prefix closed, they both represent not only finite but also infinite loopings.

Let's return now to the case of arbitrary nets, and observe that the ordering of transitions in their paths may have two causes:

1. it may be due to the sequential nature of executions; we simply have to put transitions one after another even if they can be executed independently (e.g. in parallel),
2. it may be due to the structure of the net which forces some transitions to be fired before some others.

To distinguish between these two causes in the description of net behaviors, we replace word languages by trace languages with a dependency relation between transitions defined in the following way:

$(tra_1, tra_2) : Dep.net$ iff $Neigh.tra_1 \cap Neigh.tra_2 \neq \{\}$

As we see, two transitions are dependent on each other if one of the following conditions is satisfied:

- they share a common entry, i.e. they compete in taking a token from it,
- they share a common exit, i.e., they compete in putting a token into it,
- an exit place of the one is an entry place of the other, i.e., one of them waits for the firing of the other.

Theorem 12.3.2-1 (see [72]) For any two initial paths pat_1, pat_2 of net, if $pat_1 \equiv_{Dep.net} pat_2$ then $Rf.pat_1.Ini = Rf.pat_2.Ini$ ■

By the *concurrent symbolic behavior* of net we shall mean the following trace language generated from the sequential behavior of net:

$Cbe.net = [Sbe.net]_{Dep.net}$

Since in a net any independent transitions may be fired in an arbitrary order, the following fact is easy to see:

Fact 12.3.2-1 For any net net its sequential behavior Sbe.net is Dep.net-complete, i.e.

$Sbe.net = T2L.([Sbe.net]_{Dep.net})$.

If Dep.net is full, then the net is said to be *sequential*. In that case its concurrent behavior is a sequential trace language (Sec. 12.3.1).

Fact 12.3.2-2 If net is sequential, then Sbe.net is the unique representant of Cbe.net.

Note that in a general case Sbe.net is the largest representant of Cbe.net.

Now, Petri nets may be given an executional semantics in the domain of trace languages. To do that A. Mazurkiewicz defines a universal operation of composition of two nets with disjoint sets of places. Note that the sets of transitions need not be disjoint[112]. Let

$net_i = (Place_i, Transition_i, Flow_i, ini_i)$ for $i = 1,2$

where $Place_1 \cap Place_2 = \{\}$. By *composition* of these two nets, in symbols $net_1 + net_2$ we mean the net:

net = (Place, Transition, Flow, Ini)

where

$Place = Place_1 \mid Place_2$
$Transition = Transition_1 \mid Transition_2$
$Flow = Flow_1 \mid Flow_2$
$Ini = Ini_1 \mid Ini_2$

As is easy to see, the composition of nets is associative and commutative. Besides, the dependency relation of the resulting net is the union of dependency relations of component nets.

$Dep.net = Dep.net_1 \mid Dep.net_2$

The central theorem of Mazurkiewicz's approach to Petri nets is the following:

Theorem 12.3.2-2 $Cbe.(net_1 + net_2) = Cbe.net_1 \parallel Cbe.net_2$ ■

---

[112] In fact, this is what makes the composition a non-trivial operation.

A theorem which completes our sketch of Mazurkiewicz trance theory is about a decomposition of a net into a family of atomic nets. By an *atom* of net determined by a place pla : Place we mean a one-place net defined as follows:

net_{pla} = ({pla}, Tra_{pla}, $Flow_{pla}$, Ini_{pla})

where

Tra_{pla}	= {tra | (tra, pla) : Flow **or** (pla, tra) : Flow}	transitions adjacent to pla
$Flow_{pla}$	= {(tra, pla) | pla : Exit.tra} **|** {(pla, tra) | pla : Entry.tra}	edges including pla
Ini_{pla}	= Ini ∩ {pla}	either {pla} or empty set {}

An atom of a Petri net consist of only one place plus all the edges that lead to or from this place including their transitions. Clearly

Theorem 12.3.2-3 Every Petri net is a composition of its atoms. ■

As an immediate consequence of theorems 12.3.2-2 and 12.3.2-3 we may conclude

Theorem 12.3.2-4 The concurrent behavior of a Petri net is a synchronization of concurrent behaviors of all its atoms. ■

12.3.3 Petri nets redefined

For the sake of building a denotational model of Petri nets we shall redefine the concepts of a net in a way equivalent to the former but more suitable for building an algebra of nets. In short, nets will be defined as finite sets of atoms.

Let Identifier be a set of identifiers over an alphabet including letters and digits. By an *atom* over Identifier we mean a 4-tuple which represents a marked or an unmarked net with one place:

ato : Atom = Interface x Place x Marking x Interface

where

int : Interface = FinSet.Identifier
pla : Place = Identifier
mar : Marking = {0, 1}

We assume that both, places and transitions, are identifiers. In an atom (inp-int, pla, mar, out-int) the interfaces inp-int and out-int are called respectively *input interface* and *output interface*.

By an *abstract Petri net* over Identifier we mean any finite set of atoms with mutually different places. We call them "abstract nets" since in Sec. 12.3.4 we introduce concrete nets. Formally, they are, of course, different from nets of Sec. 12.3.2. We define two domains which will become carriers of the future *algebra of abstract nets*

abn : AbsNet = AbstractNet.Identifier
int : Interface = FinSet.Identifier

where AbstractNet is a domain constructor analogously as ⟹ or ⟼.

An abstract net abn-1 is said to be a *subnet* of abstract net abn-2 if abn-1 ⊆ abn-2[113]. Abstract nets abn-1 and abn-2 are said to be *separated* if their sets of places and of transitions are disjoint. The constructors of the algebra of abstract nets are the following:

build-empty-interface	:	⟼ Interface
add-to-interface	: Identifier x Interface	⟼ Interface
build-unmarked-atom	: Interface x Identifier x Interface	⟼ AbsNet
build-marked-atom	: Interface x Identifier x Interface	⟼ AbsNet

[113] Note that this notion of a subnet may be different from such notions defined by other authors.

assemble-abs-nets : AbsNet x AbsNet ↦ AbsNet

We skip obvious definitions of the first four constructors. To define the last one we introduce an auxiliary function of prefixing places of nets with digits '1' and '2':

prefix : AbsNet x {'1', '2'} ↦ AbsNet
prefix.((inp, pla, mar, out), p) = (inp, p • pla, mar, out) where p : {'1', '2'}
prefix.({ato-1,…,ato-n}, p) = {prefix.(ato-1, p),…,prefix.(ato-n, p)}

The operation of *assemblage* of two abstract nets is defined in the following way:

assemble-abs-nets.(abn-1, abn-2) = prefix.(abn-1, '1') | prefix.(abn-2, '2')

Now, due to the Theorem 12.3.2-3, we can claim that for every Petri net as defined in Sec. 12.3.2 there exist an "equivalent" abstract net, and vice versa. By "equivalent nets" we mean that their corresponding graphs are isomorphic.

Having defined an algebra of abstract nets we have to define three functions which describe their symbolic behaviors:

Sbe : AbstractNet.Identifier ↦ Lan.Identifier sequential symbolic behavior
Cbe : AbstractNet.Identifier ↦ TraLan.Identifier concurrent symbolic behavior
Dep : AbstractNet.Identifier ↦ FinSet.(Identifier x Identifier) dependency relation

We denote them by the same symbols as in Sec. 12.3.2, and, of course, they must correspond — in an obvious sense — to these functions. All three function will be defined by structural induction starting from two basic cases of atomic nets:

ato = (inp, pla, 0, out) and ato = (inp, pla, 1, out).

In this case (cf. Fig. 12.3-1):

Sbe.ato = $(\text{inp x out})^{c*}$ x (inp | {()}) Sbe.ato = $(\text{out x inp})^{c*}$ x (out | {()})[114]
Dep.ato = (inp | out) x (inp | out) Dep.ato = (inp | out) x (inp | out)
Cbe.ato = $[\text{Sbe.ato}]_{\text{Dep.ato}}$ Cbe.ato = $[\text{Sbe.ato}]_{\text{Dep.ato}}$

In the case of atomic nets the dependency relations are full, and therefore their concurrent behaviors are sequential.

Note that a sequential behavior of a net is a set of sequence of identifiers representing transitions. Consequently it may be regarded as a language over an alphabet Identifier. Since identifiers are themselves sequences of letters, in the definition of Sbe.ato we have used Cartesian product and Cartesian star rather than concatenation and star of languages. Now, given two arbitrary abstract nets abn-1 and abn-2 and their composition

abn = assemble-abs-nets.(abn-1, abn-2)

we set (cf. Theorem 12.3.2-2)

Cbe.abn = Cbe.abn-1 ‖ Cbe.abn-2
Sbe.abn = T2L.(Cbe.abn)
Dep.abn = Dep.abn-1 | Dep.abn-2

We recall that by the definition of the synchronization operation ‖, traces of Cbe.abn are built over Dep.abn. Note now that the following equations are satisfied

Sbe.abn = T2L.(Cbe.abn-1 ‖ Cbe.abn-2) = by the definition of Cbe

[114] Here we half formally assume that Cartesian product is associative which means that $(\text{inp x out})^{c*}$ and $(\text{inp x out})^{c*}$ are set of sequences of identifiers, rather than set of sequences of pairs of identifiers.

T2L.([Sbe.abn-1]$_{\text{Dep.abn-1}}$ ‖ [Sbe.abn-2]$_{\text{Dep.abn-2}}$) = by Theorem 12.3.1-4
T2L.([Sbe.abn-1 ‖ Sbe.abn-2]$_{\text{Dep.abn-1 | Dep.abn-2}}$)

This means that the sequential behavior of an assemblage of two nets is expressible as a combination of the sequential behaviors of the component nets.

Fact 12.3.3-1 For any two abstract nets abn-1 and abn-2

Sbe.(assemble-abs-nets.(abn-1, abn-2)) = T2L.([Sbe.abn-1 ‖ Sbe.abn-2]$_{\text{Dep.abn-1 | Dep.abn-2}}$).

It is worth recalling in this context that (cf. Fact 12.3.2-1) the equality:

Cbe.(assemble-abs-nets.(abn-1, abn-2)) =
[Sbe.(assemble-abs-nets.(abn-1, abn-2))]$_{\text{Dep.abn-1 | Dep.abn-2}}$

and the fact that Sbe.(assemble-abs-nets.(abn-1, abn-2)) is Dep.abn-1|Dep.abn-2 – complete.

Consider as an example an abstract net consisting of two interleaving loops in Fig. 12.3-2.

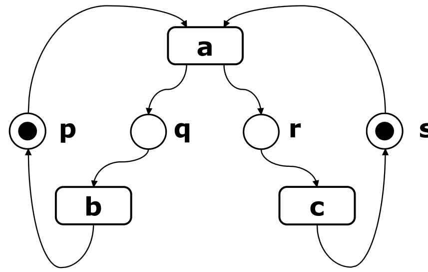


Fig. 12.3-2 A net with two interleaving transitions

This net may be regarded as an assemblage of four atomic nets, where a stands for {(a)}, and similarly for other transitions:

abn-1 = (b, p, 1, a), Sbe.abn-1 = (ab)* (a | ()), Dep.abn-1 = {a, b}^{c2}
abn-2 = (b, q, 0, a), Sbe.abn-2 = (ab)* (a | ()), Dep.abn-2 = {a, b}^{c2}
abn-3 = (a, r, 0, c), Sbe.abn-3 = (ac)* (a | ()), Dep.abn-3 = {a, c}^{c2}
abn-4 = (c, s, 1, a), Sbe.abn-4 = (ac)* (a | ()), Dep.abn-4 = {a, c}^{c2}

Since the dependency relations of all these nets are full, they are sequential nets, and their concurrent behaviors are sequential trace languages. The same concerns the assembled nets

abn-12 = assemble-abs-nets.(abn-1, abn-2), Dep.abn-12 = {a, b}^{c2}
abn-34 = assemble-abs-nets.(abn-3, abn-4), Dep.abn-34 = {a, c}^{c2}

In this case

Cbe.abn-12 = [(ab)* (a | ())] ‖ [(ab)* (a | ())] = by Theorem 12.3.1-4
[((ab)* (a | ())) ‖ ((ab)* (a | ()))] =
[(ab)* (a | ())]

Sbe.abn-12 = (ab)* (a | ()) by Fact 12.3.2-2 since abn-12 is sequential

Analogously

Cbe.abn-34 = [((ac)* (a | ()))]
Sbe.abn-34 = (ac)* (a | ()))

Assembling these two sequential nets we get a non-sequential net:

abn = assemble-abs-nets.(abn-12, abn-34) with Dep.abn = {a, b}^{c2} | {a, c}^{c2},
ind.Dep.abn = {(b, c), (c, b)}

For this net

Cbe.abn = [(ab)* (a | ())] ‖ [(ac)* (a | ())] =

[((ab)* (a | ())) ‖ ((ac)* (a | ()))]
Sbe.abn = T2L.[((ab)* (a | ())) ‖ ((ac)* (a | ()))]

12.3.4 Petri nets with data flow

Petri nets defined so far describe control flows of concurrent programs. To enrich them with a mechanism of data flow we assign bundles of computations to transitions. Let State be an arbitrary set of states. At this stage of our investigations we do not need to assume anything about states. By *transition dictionaries* we mean mappings:

tdi : TraDic = Identifier ⇒ Bun.State

which assign bundles of states to transitions. By a *concrete net* we mean a pair consisting of an abstract net and a transition dictionary:

cne : ConNet = AbsNet x TraDic

Concrete net (abn-1, tdi-1) is called a *subnet* of a concrete net (abn-2, tdi-2) if

abn-1 ⊆ abn-2 and tdi-1 ⊆ tdi-2.

A concrete net (abn, tdi) is said to be *well formed* if all transitions of abn belong to the domain of tdi. By a *concrete execution* of a well-formed net (abn, tdi) we mean a sequence of pairs

((tra-1, com-1)…,(tra-n, com-n)) where com-i : $State^{c+}$

such that

(1) (tra-1,…,tra-n) : Sbe.abn
(2) com-i : tdi.tra-i for i = 1;n
(3) com-1 ● … ● com-n ≠ ()

Condition (1) expresses the fact that a concrete execution can happen symbolically, and condition (3) — that it can happen semantically.

We say that a concrete execution *starts in a set of states* con ⊆ State if the first state of com-1 belongs to con. By

coe : ConExe.cne = {((tra-1, com-1)…,(tra-n, com-n)) | (1), (2), (3) satisfied}

we shall denote the set of all concrete executions of a well-formed concrete net cne. By a *denotational behavior* of a well-formed concrete net (abn, tdi) we mean a bundle defined in the following way:

Dbe.(abn, tdi) = {com-1 ● … ● com-n | (∃ (tra-1,…,tra-n) : Sbe.abn)
((tra-1, com-1)…,(tra-n, com-n)) : ConExe.(abn, tdi)}

We say that a concrete executions is *a prefix* of another one, in symbols

((tra-1, com-11),…,(tra-1, com-1n)) ⊏ ((tra-21, com-21),…,(tra-2m, com-2m)) (*)

iff

n < m and ((tra-1, com-11),…,(tra-1, com-1n)) = ((tra-21, com-21),…,(tra-2n, com-2n))

Of course, ConExe.(abn, tdi), similarly to Sbe.abn is prefix closed. Note that by the definition of a concrete execution if (*) holds then

com-21 ● … ● com-2m ≠ ()

and therefore

com-11 ● … ● com-1n ⊏ com-21 ● … ● com-2m or

com-11 ● … ● com-1n = com-21 ● … ● com-2m

The equality may take place if all computations com-2(n+1),…,com-2m are appropriate one-element computations.

By a *concrete branch* of a concrete net we mean a finite or infinite sequence of concrete executions of this net such that each of them, except the last one, if it exists, is a prefix of the next one. We say that a concrete execution is *terminal* in a concrete net if it is not a prefix of any other concrete execution of this net.

By an *assemblage* of two concrete nets we mean a concrete net whose abstract component is an assemblage of abstract nets and the dictionary is an overwriting of dictionaries:

assemble -con-nets : ConNet x ConNet ⟼ ConNet

assemble -con-nets.((abn-1, tdi-2), (abn-2, tdi-2)) =
  (assemble-abs-nets.(abn-1, abn-2), tdi-1 ♦ tdi-2)).

By a *cocoon*[115] we mean a tuple (in-tra, (abn, tdi), out-tra) such that

1. (abn, tdi) is a well-formed concrete net,
2. in-tra and out-tra are transitions of abn,
3. in-tra has no entrance places (orphan) and out-tra has no exit places (widow) in abn,
4. in-tra is the only orphan and out-tra is the only widow of abn,
5. initial marking of abn is empty.

in-tra

Note that by the properties 4. and 5. all sequential executions of abn start with in-tra.

By the *flow* of a cocoon con = (in-tra, (abn, tdi), out-tra) we mean a bundle generated by symbolic executions of concrete net (abn, tdi) between input and output transitions, i.e. all executions at all (!) which terminate in out-tra.

Flow.(in-tra, abn, tdi, out-tra) =

  {com-1 ● … ● com-n | (∃ (tra-1,…,tra-n) : Sbe.abn)
      tra-n = out-tra **and**
      ((tra-1, com-1)…,(tra-n, com-n)) : ConExe.(abn, tdi)}

Of course

Flow.(in-tra, abn, tdi, out-tra) ⊆ Dbe.(abn, tdi).

Let con-1, con-2 ⊆ State. A cocoon coc is said to be *strongly totally correct*[116] or simply *correct* wrt a precondition con-pr and a postcondition con-po, in symbols

pre con-pr; coc **post** con-po

if

1. there is no infinite concrete branch that starts with a state in con-pr; no semantic livelock,
2. all terminal concrete execution that start with a state in con-pr terminate with transition out-tra in a state in con-po; no semantic deadlock.

Of course, **pre** con-pr; coc **post** con-po iff (see Sec. 12.2.3)

---

[115] We use this word since a cocoon may be said to be a thread with a skein in the middle.

[116] In the case of deterministic programs total correctness means that the (unique) execution of the program terminates. The clean total correctness means that the execution terminates without abortion, and the strong total correctness of nondeterministic programs — that all executions terminate without abortion.

con-p ⊆ Flow.coc ■ con-po

Note that the freeness of semantic livelock and deadlock of a cocoon does not imply that the corresponding abstract net if free of livelock or deadlock respectively.

Poproszę kolegów o przykład ???

Let a concrete net cne-1 be a subnet of a concrete net cne-2. By a *projection* of a concrete executions coe of cne-2 on a subnet cne-1, in symbols

pro.abn-1.coe

we mean the result of removing from coe all pairs (tra, com) such that tra is not a transition of con-1. For a formal definition of a similar function see Sec. 12.3.1.

Consider two concrete nets cne-i = (abn-i, tdi-i) for i = 1,2, two corresponding sets of states (conditions) con-i ⊆ State for i = 1,2, and let

cne = assemble-con-net.(cne-1, cne-2)

We say that these nets are *semantically independent* for corresponding conditions, if

(1) abn-1 and abn-2 are separated and

(2) for any concrete execution of their composition

coe : ConExe.(assemble-con-net.(cne-1, cne-2))

if pro.abn-i.coe starts with a state in con-i then

pro.abn-i.coe : ConExe.cne-i for i = 1,2.

Condition (2) says that every projection of coe on abn-i that starts in con-i could happen as an independent execution of cne-i.

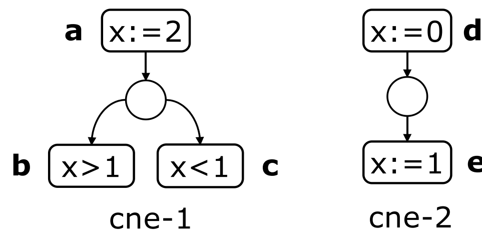


Fig. 12.3-3 Two concrete dependent nets

Consider as an example two concrete nets in Fig. 12.3-3, and assume their following properties:

1. both nets operate on states that are mappings from identifiers x, y to arbitrary numbers, i.e.

   sta : State = {x, y} ⟹ Number

2. a common precondition con for both nets claims that variables x and y have been declared to be of type integer,
3. bundles assigned to transitions are the following sets of pairs of states (they are functional bundles):
   a. {(sta, sta[x/2]) | sta : State}
   b. {(sta, sta) | sta : State **and** sta.x > 1}
   c. {(sta, sta) | sta : State **and** sta.x < 1}
   d. {(sta, sta[x/0]) | sta : State}
   e. {(sta, sta[x/1]) | sta : State}

The following sequence is an example of a concrete execution of the assemblage of cne-1 and cne-2 where sta is an arbitrary state which satisfies con:

(a, (sta, sta[x/2]), (d, (sta[x/2], sta[x/0]), (c, (sta[x/0], sta[x/0])), (e, (sta[x/0], sta[x/1]))

The projection of this execution on cne-1 is the following

(a, (sta, sta[x/2]), (c, (sta[x/0], sta[x/0]))

and it is not a concrete execution of net-1 since

(sta, sta[x/2]) ● ((sta[x/0], sta[x/0])) = ()

If in transit d we set y:=0 then our two concrete nets become semantically independent. Note that our nets become independent also if in d we set x:=2.

12.4 Building a language of concurrent programs

12.4.1 General assumptions about the language

In this section we propose a sketchy idea of how to extend a denotational model of a languages of sequential programming such as, e.g., **Lingua** or **Lingua-SQL**, to a denotational model covering concrete Petri nets. Similarly as in the case of **Lingua-SQL** we shall restrict our investigations to an algebra of denotations.

Let's assume at the beginning that **AlgDen** is a given algebra to be extended, and that the denotations of instructions, declarations and programs in this algebra are bundles of computations. The extended algebra **AlgDenCCP** is created from the former by adding to it the following new carriers:

abn	: AbsNet	= PetriNet.Identifier	Petri nets
int	: Interface	= FinSet.Identifier	interfaces
tdi	: TraDic	= Identifier ⇒ InsDen	transition dictionaries
cne	: ConNet	= AbsNet x TraDic	concrete nets

and the corresponding constructors

build-single-interface	: Identifier	⟼ Interface
add-to-interface	: Identifier | Interface	⟼ Interface
build-unmarked-atom	: Interface x Identifier x Interface	⟼ AbsNet
build-marked-atom	: Interface x Identifier x Interface	⟼ AbsNet
assemble-abs-nets	: AbsNet x AbsNet	⟼ AbsNet
build-dictionary	: Identifier x TraDen	⟼ TraDic
add-to-dictionary	: Identifier x TraDen x TraDic	⟼ TraDic
create-con-net	: AbsNet x TraDic	⟼ ConNet
asamble-con-nets	: ConNet x ConNet	⟼ ConNet
encapsulate-con-net	: ConNet	⟼ InsDen

An instruction in the new algebra may be an instruction in the former sense or an *encapsulated concrete net*. In turn, transitions may carry arbitrary instructions.

The definitions of all new constructors but the last one are trivial. The last constructor given a concrete net returns a bundle of computations that belongs to the domain of instruction denotations:

encapsulate-con-net.(net, tdi) = Dbe.(net, tid)

Note that this constructor "forgets", in a sense, the structure of the argument net replacing it by a corresponding bundle. This bundle may appear later either as an argument of a constructor of instruction denotations in the basic algebra or a transition denotations assigned to a dictionary.

12.4.2 A case study of a structured constructor

New constructors described in Sec. 12.4.1 allow to build concrete nets with arbitrary net structures. In the world of sequential programs this "flexibility" may be compared to allowing arbitrary flowchart programs built by **goto**'s. It is a well-known fact that such solution leads to programs that are hard to understand and even harder to prove correct.

A solution of this problem for sequential programming consists in restricting the control structures of instructions by structural constructors. For each such constructor we can create a dedicated proof rule, and in our case — a construction rule that guarantees correctness. In this way, instead of struggling with correctness proofs of "arbitrary programs" we are building a limited class of programs but their correctness proofs are implicit in the processes of their construction.

Here we propose an analogous solution for concurrent programming. However, we do not attempt to provide any "complete collection" of constructors for building nets since this would lead to a profound research certainly going beyond the scope of our book. We shall limit our attention to only one of a structured constructor of concurrent programs that given two critical sections and two corresponding non-critical sections builds a net with Dijkstra's semaphores synchronizing the cooperation of sections. The expected correctness property of such a net, which we shall call *adequacy*, is described as follows:

1. the net is deadlock free,
2. each of its sections is livelock free,
3. critical sections are mutually excluded, i.e. their transitions can't interleave,
4. each non-critical section is excluded for its critical section,
5. after an execution of a non-critical section its corresponding critical section will be executed — a non-starvation property.

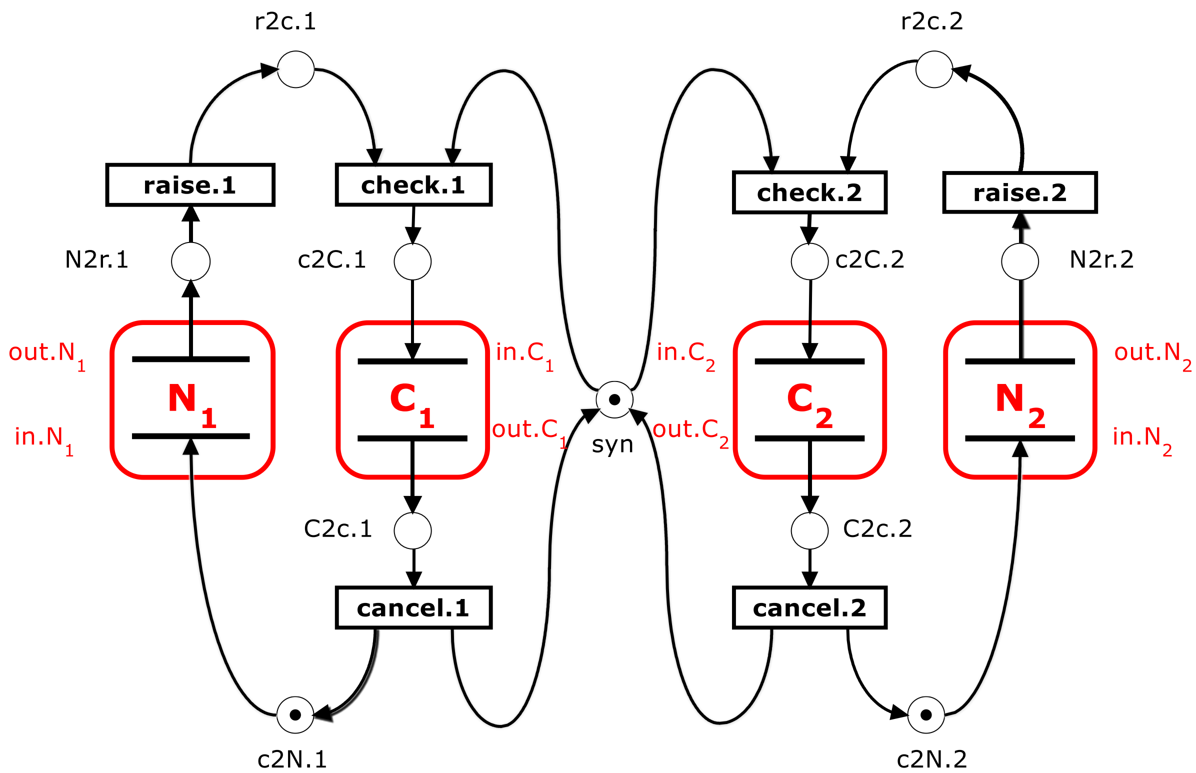


Fig. 12.4-1 Critical sections with semaphores

We start from building such a net in using constructors defined in Sec. 12.4.1. The structure of the abstract part of our target net is shown in Fig. 12.4-1. It consists of four subnets representing two critical sections, and two non-critical sections (all four in red) plus one synchronizing net (in black), that we shall call SynNet, including semaphores.

The target concrete net, let's call it SemNet, may be built as a composition of these five nets:

SemNet = assemble-con-nets.(SynNet,
assemble-con-nets.((N_1, tdi-n_1),
assemble-con-nets.((C_1, tdi-com-1),
assemble-con-nets.((N_2,, tdi-n_2), (C_2, tdi-c_2)))))

Now, the question is, what do we have to assume about our five nets to make sure that SemNet is adequate. We start from assuming that the four sections constitute mutually separated cocoons:

non-cri.1 = (in.N_1, (N_1, tdi-nc.1), out.N_1)
critical.1 = (in.C_1, (C_1, tdi-cr.1), out.C_1)
non-cri.2 = (in.N_2, (N_2, tdi-nc.2), out.N_2)
critical.2 = (in.C_2, (C_2, tdi-cr.2), out.C_2)

Next we assume that the abstract part AbsSynNet of the synchronization net SynNet is a composition (a set) of the following atomic nets:

(cancel.1, c2N_1, 1, in.N_1) (cancel.2, c2n.2, 1, int.N_2)
(out.N_1, N2r.1, 0, raise.1) (out.N_2, n2r.2, 0, raise.2)
(raise.1, r2c.1, 0, check.1) (raise.2, r2r.2, 0, check.2)
(check.1, c2C.1, 0, in.C_1) (check.2, r2c.2, 0, int.N_2)
(out.C_1, C2c.1 0, cancel.1) (out.C_2, C2c.2, 0, cancel.2)
({cancel.1, canclel.2}, syn, 1, {check.1, check.2)

For simplicity we write transition for {transition}, and we give mnemotechnical names to places, e.g. c2N_1 is read as "from cancel to N_1".

To assure that SynNet realizes the mechanism of semaphores we have to assign an appropriate dictionary to this net. To do that we assume that queue is a variable which stores a FIFO queue of digits '1' and '2', and we define three typical operations on queues:

first.(q_1,…,q_n) = q_1, first.() = ? (we assume that first is a partial function)
put.q.(q_1,…,q_n) = (q_1,…,q_n, q)
cut.(q_1,…,q_n) = (q_2,…,q_n), cut.() = ()

The dictionary tdi-sta-n of SynNet is defined in the following way:

tdi-sn.cancel.1 = {(sta, sta[queue/cut.(sta.queue) }
tdi-sn.raise.1 = {(sta, sta[queue/put.'1'.(sta.queue)]}
tdi-sn.check.1 = {(sta) | first.(sta.queue) = '1', }
tdi-sn.cancel.2 = {(sta, sta[queue/cut.(sta.queue) }
tdi-sn.raise.2 = {(sta, sta[queue/put.'2'.(sta.queue)]}
tdi-sn.check.2 = {(sta) | first.(sta.queue) = '2', }

Hence SynNet = (AbsSynNet, tdi-sn). Now, we have to formulate assumptions sufficient for SemNet to be adequate. In the first place cocoons must be free of deadlocks and livelocks. Precisely speaking we have to assume that they are correct wrt some pre- and post-conditions. These conditions should also guarantee that semaphores will work as expected. This leads us to the following assumptions about SemNet:

There exist sets of states (pre- and post-conditions):
nc-pre.i and nc-post.i for i = 1,2,
cr-pre.i and cr-post.i for i = 1,2
such that:

a. **pre** nc-pre.i; non-cri.i **post** nc-post.i for i = 1,2,
b. **pre** cr-pre.i; critical.i **post** cr-post.i for i = 1,2,
c. for i ≠ j, non-cri.i is semantically independent of non-cri.j and critical.j under corresponding preconditions,
d. for i ≠ j, critical.i is semantically independent of non-cri.j and critical.j under corresponding preconditions,

e. cr-post.i ● tdi-sn.cancel.i ⊆ nc-pre.i for i = 1,2
f. nc-post.i ● tdi-sn.raise.i ● tdi-sn.check.i ⊆ cr-pre.i for i = 1,2
g. cr-post.i ● tdi-sn.cancel.i ● tdi-sn.check.j ⊆ cr-pre.j for i ≠ j
h. the value of queue is neither modified nor used by the instructions of cocoons.

Under these assumptions we can claim more about SemNet than just the freeness of deadlock, livelock and no-starvation. We can prove that for every concrete execution of SemNet that starts either from non-cri.1 in a state in nc-pre.1 or from non-cri.2 in a state in nc-pre.2, the following properties are satisfied:

1. the executions is free of deadlock and livelock,
2. critical sections critical.1 and critical.2 are mutually excluded, i.e. only one of them may be run at a time,
3. sections non-cr.i and critical.i are mutually excluded for i = 1,2,
4. after the execution of non-cr.i an execution of critical.i will eventually happen for i = 1,2 (non-starvations),
5. after the execution of non-cr.i its next execution is possible only after the execution of critical.i, for i = 1,2,
6. after the execution of critical.i its next execution is possible only after the execution of non-cr.i, for i = 1,2,
7. transitions of non-cr.i may arbitrarily interleave with transitions of critical.j and non-cri.j for i ≠ j.

At this point we have constructed a net with semaphores using just one constructor assemble-con-net but our construction does not meet, the following expectations of M. Ben-Ari (see [13] p. 145)

(...) the semaphore is a low-level primitive because it is unstructured. If we were to build a large system using semaphores alone, the responsibility for the correct use of the semaphores would be diffused among all the implementors of the system. If one of them forgets to call signal(S) *after a critical section, the program can deadlock and the cause of the failure will be difficult to isolate.*

In fact, our solution bases on the assumption that our net constructor receives SynNet as an argument which means that semaphores are built into SemNet by a programmer. What we would like to have is a constructor which given our four cocoons builds SemNet, which means that it builds SynNet which satisfies conditions from 5 to 8.

Here the critical condition is, of course, 8. To satisfy this condition we have to indicate an identifier queue with the assumed properties. The simplest constructive solution seems to be the selection of an identifier which is declared in states of nc-pre.1 | nc-pre.2 and does not appear syntactically in the instructions of our cocoons. Such a solution is, however, not feasible, since, informally speaking, the information carried by our cocoons is not sufficient to select such an identifier. The only solution that we can see at the moment is to assume that this identifier is given as an argument of the future constructor, i.e., is given by a programmer. In that case our expected constructor may be the following:

```
semaphore-net.(non-cri.1, critical.1, non-cri.2, critical.2, queue) =
  let
    (in.N1,  (N1, tdi-nc.1),   out.N1) = non-cri.1
    (in.C1,  (C1, tdi-cr.1),   out.C1) = critical.1
    (in.N2,  (N2, tdi-nc.2),   out.N2) = non-cri.2
    (in.C2,  (C2, tdi-cr.2),   out.C2) = critical.2
    SynNet = defined as above
  true ➔ assemble-con-nets.( SynNet,
             assemble-con-nets.( (N1, tdi-n1),
               assemble-con-nets.( (C1, tdi-com-1),
                 assemble-con-nets.( (N2,, tdi-n2), (C2, tdi-c2) ) ) ) )
```

This definition looks similar to that of SemNet however now SynNet is not "given ahead", but is built out of the arguments of semaphore-net.

Since SynNet is not an argument of our constructor, its elements should not appear in the prerequisites of our adequacy rule. Formally they do, but practically they don't, since if we assume that the pre- and post-conditions do not depend on variable queue, then 5., 6. and 7. may be reduced to inclusions (i.e. implications):

cr-post.i ⊆ nc-pre.i for i = 1,2
nc-post.i ⊆ cr-pre.i for i = 1,2
cr-post.i ⊆ cr-pre.j for i ≠ j

Our last remark concerns the way we can view our adequacy rule. If we see it as a proof rule, it is rather poor, since its prerequisites are pretty strong. However, in our approach we aim at constructing correct programs rather than proving (arbitrary) programs correct. It practically means that before we apply our constructor we have to build its arguments in such a way that they have expected properties.

13 WHAT REMAINS TO BE DONE

Even though the book is already of a considerable volume, the majority of subjects have only been sketched. What remains to be done is enough for a few more books and also as a research and development area for many researchers and developers. Below we suggest a preliminary list of subjects which is certainly not complete. It considers both research problems as well as programming (implementational) tasks.

13.1 Computer-aided program development

On a theoretical ground, our method of the development of correct metaprograms offers a collection of mathematical tools dedicated to developing and proving the correctness of metaprograms, i.e., theorems of the form:

pre prc **:** spr **post** poc.

It is fairly evident that the use of our tools in practice requires an assistance of a software system. Below we share some very preliminary thoughts about such a system. In our opinion it might consist of three main modules:

1. an intelligent editor supporting the writing of metaprograms in **Lingua-V** using Visual Studi Code,
2. a dedicated theorem-prover supporting the application of program-construction rules,
3. an implementation of **Lingua**, i.e., a parser and an interpreter or compiler.

The first attempt to build an implementation of **Lingua** (without objects) has been undertaken by a small group of two teachers (me and Aleksy Schubert) and three of our students at the Department of Mathematics, Informatics, and Mechanics of Warsaw University during the Spring Semester of the year 2020 (see [38]). To tell the truth, my role was limited in this case to checking if the developed implementation was compatible with the model of **Lingua,** as described in this book. The bulk of the work was done by Aleksy and the students. The programming language of implementation was OCaml.

The editor of programs should support two types of tasks:

1. keeping derived metaprograms syntactically correct,
2. keeping derived metaprograms statically correct[117], e.g., no identifier should be declared twice in one program, or actual parameters of a procedure should be statically compatible with formal parameters.

The dedicated theorem-prover should assist programmers in:

1. proving metaimplications appearing above the lines in program-construction rules,
2. keeping and updating a library of currently proved theorems.

In turn the library of theorems should include the following sublibraries:

1. a library of system-dependent theorems:
    a. basic theorems about data, values, types and denotations appearing in the model of **Lingua-V**,
    b. currently proved program-construction rules,
    c. currently proved concrete correct metaprograms,
2. a library of program-dependent theorems for the currently developed program:

---

[117] This concept is related to so called “static semantics” that describes such properties of syntax that can’t be described by grammars and therefore can’t be checked by parsers.

  a. perpetual conditions in the current program,
  b. hereditary conditions associated with current cuts of this programs,

Whereas part 1. of this library would be modified only occasionally, part 2. must be updated in each step of program development.

As we pointed out in Sec. 9.4.1, the development of a metaprogram may be split into a sequence of steps. In each step, given some earlier developed metaprograms we create a new metaprogram by means of one of our construction rules. The application of a rule may require proving some "local lemmas" required by the rule, e.g., a metaimplication:

con1 ⇨ con2.

Such a metaimplication will be always proved in the context of the current content of our library, which formally means that our prover will prove the truth of the following metaformula (cf. Sec. 9.3.2):

con1 ⇨ con2 **whenever** imm-con **and** per-con **and** her-con

where

imm-con is the conjunction of all immunizing condition from part 1.a of the library,
per-con is the conjunction of all perpetual conditions from part 2.a of the library,
her-con is the conjunction of all hereditary conditions from part 2.b of the library.

Similarly, whenever a programmer will develop a metaprogram of the form

```
pre prc:
   spr
post poc
```

the prover will elaborate

```
pre prc and imm-con and per-con and her-con :
   spr
post poc and imm-con and per-con and her-con
```

In each step of program development the part 2. of the library may be updated.

13.2 Computer-aided language design

The module dedicated to supporting the design of programming languages in our framework might include the following components:

1. An intelligent editor supporting the writing of the definitions of the algebra of denotations:
   a. the definitions of the carriers of the algebra, i.e., the denotational domains; here the editor might check if domain equations do not include a not acceptable recursion,
   b. he definitions of the constructors of the algebra.
2. A system supporting syntax design:
   a. a generator of an equational grammar of an abstract syntax for given definition of constructors from 1.b,
   b. a system supporting the development of an equational grammar of concrete syntax out of abstract syntax,
   c. a system supporting the generation of an equational grammar of colloquial syntax out of concrete syntax,
   d. a system supporting the generation of the definition of a restoring transformation.
3. A system supporting the writing of a definition of semantics.

4. A system supporting the development of a parser of the designed language out of the definition of its syntax.
5. A system supporting the development of an interpreter of the designed language out of the definition of its syntax and semantics.
6. A system supporting the development of a compiler of the designed language out of the definition of its syntax and semantics.

Although the tasks of writing a programmer's interface and a language designer's interface are in principle independent, one may think of two possible scenarios of building them:

1. first a version of **Lingua** is developed without using an interface of a language designer and the latter is then written in **Lingua**,
2. a language-designer interface is written in one of existing languages, and later the full definition of **Lingua** along with its implementation is developed in this system.

13.3 Techniques of writing user manuals

Denotational models should provide an opportunity to revise current practices seen in the manuals of programming languages. On the one hand, new practices should be based on denotational models, but on the other, do not assume that today's readers are experts in this field. A manual should, therefore, provide some basic knowledge and notation needed to understand the definition of a programming language written in a new style. At the same time, we firmly believe that it should be written for professional programmers rather than amateurs. In our opinion, the role of a manual is not to teach programming skills. Such textbooks are, of course, necessary, but they should teach the readers what programming is about rather than the technicalities of a concrete language. Unfortunately, the current practice usually contradicts these principles.

13.4 Programming experiments

For our idea of correct-program development to be noticed by the IT community, some convincing applications must be shown. In our (preliminary) opinion, an adequate field for such applications may be microprogramming because:

1. microprograms contain a relatively small number of the lines of code,
2. their correctness is highly critical,
3. highly critical is also the memory- and time-optimisation of such programs.

13.5 Building a community of Lingua supporters

Our methods of designing programming languages and constructing programs may be assessed positively or negatively, but one thing seems evident — they are pretty far from current practices. The book offers a far-going change, and such changes always provoke groups of opponents and supporters. The former should be convinced, and the latter must be strengthened. And, of course, one has to start from the first task.

To realize that task, one has to give the potential supporters some — may be very simple — still sufficiently practical version of **Lingua.** An alternative may consist of encouraging them to build their version. The first solution seems somewhat unrealistic since it would require finding an investor for a strange and utterly unknown product. The other way is that an experimental **Lingua** is built by volunteers and for volunteers, as in the case of Linux, Joomla!, or MySQL. However, such a product, although freely available, should not be open-source since this might lead to mathematically incorrect solutions and consequently to unsound program-construction rules.

Therefore, the **Lingua** builders community must elaborate rules of accepting new members and giving them the right to join implementation teams.

14INDICES AND GLOSSARIES

14.1 References

[1] Aalst Wil van der, Hee Kees van, *Workflow management: models, methods, and systems (Cooperative Information Systems)*, MIT Press 2004

[2] Ahrent Wolfgang, Beckert Bernhard, Bubel Richard, Hähnle Reiner; Schmitt Peter H., Ulbrich Mattias (Eds.), *Deductive Software Verification — The KeY Book; From Theory to Practice,* Lecture Notes in Computer Science 10001, Springer 2016

[3] Aho A.V., Ullman J.D., *The Theory of Parsing, Translation, and Compilation, volume 1, Parsing*, Prentice-Hall, Englewood Cliffs, NJ 1972

[4] Apt K.R., *Ten Years of Hoare's Logic: A Survey - Part 1*, ACM Trans. Program. Lang. Syst. 3(4): 431-483 (1981)

[5] Apt Krzysztof R., Olderog Ernst-Rüdiger, Fifty years of Hoare's Logic, Springer 2020

[6] Apt Krzysztof R., Boer (de) Frank, S., Olderog Ernst-Rüdiger, *Verification of Sequential and Concurrent Programs*, Third, Extended Edition, Springer 2020

[7] Backus J.W., Bauer F.L., Green J., Katz C., McCarthy J., Naur P. (Editor), Perlis A.J., Rutishauser H., Samelson K., Vauquois B., Wegstein J.H., Van Wijngaarden A., Woodger M., *Report on the algorithmic language ALGOL 60*, Numerische Mathematik 2, 106--136 (1960)

[8] Bakker Jaco (de), *Mathematical Theory of Program Correctness*, Prentice/Hall International 1980

[9] Banachowski Lech, *Bazy danych. Tworzenie aplikacji*, Akademicka Oficyna Wydawnicza PLJ, Warszawa 1998

[10] Banachowski Lech, Kreczmar Antoni, Mirkowska Grażyna, Rasiowa Helena, Salwicki Andrzej, *An introduction to Algorithmic Logic — Metamathematical Investigations of Theory of Programs*, T. 2: Banach Center Publications. Warszawa PWN, 1977, s. 7-99, series: Banach Center Publications, vol.2

[11] Barringer H., Cheng J.H., Jones C.B., *A logic covering undefinedness in program proofs*, Acta Informatica 21 (1984), pp. 251-269

[12] Bekić Hans, *Definable operations in general algebras and the theory of automata and flowcharts* (manuscript), IBM Laboratory, Vienna 1969

[13] Ben-Ari Mordechai, *Principles of Concurrent and Distributed Programming*, second edition, Addison-Wesley 2006

[14] Binsbergena L. Thomas van, Mosses Peter D., Sculthorped C. Neil, *Executable Component-Based Semantics*, Preprint submitted to JLAMP, accepted 21 December 2018

[15] Bjørner Dines, Jones B. Cliff, *The Vienna development method: The metalanguage*, Prentice-Hall International 1982

[16] Bjørner Dines, Oest O.N. (ed.), *Towards a formal description of Ada*, Lecture Notes of Computer Science 98, Springer Verlag 1980

[17] Blikle Andrzej, *Automaty i gramatyki — wstęp do lingwistyki matematycznej*, (Automata and Grammars — An Introduction to Mathematical Linguistics) PWN 1971

[18] Blikle Andrzej, *Algorithmically definable functions. A contribution towards the semantics of programming languages*, Dissertationes Mathematicae, LXXXV, PWN, Warszawa 1971

[19] Blikle Andrzej, *Nets; complete lattices with a composition*, Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronomy Phys. 19 (1971), pp. 859-863

[20] Blikle Andrzej, *Equational Languages*, Information and Control, vol.21, no 2, 1972

[21] Blikle Andrzej, *An algebraic approach to programs and their computations*, Mathematical Foundations of Computer Science II (Proc. Symp. High Tatras 1973, High Tatras 1973 pp. 3 – 8

[22] Blikle Andrzej, *Proving programs by sets of computations*, Mathematical Foundations of Computer Science III (Proc. Symp. Warsaw-Jadwisin 1974), Lecture Notes in Computer Science, Vol. 28, Springer Verlag, Heidelberg 1975, pp. 313 – 3558.

[23] Blikle Andrzej, *Proving programs by δ-relations*, Formalization of Semantics of Programming Languages and Writing Compilers, (Proc. Symp. Frankfurt am Oder 1974). Elektronishe Informationesverarbeitung und Kybernetik, 11 (1975), pp. 267 – 274

[24] Blikle Andrzej, *An analysis of programs by algebraic means*, Mathematical Foundations of Computer Science, Banach Center Publications, vol.2, Państwowe Wydawnictwa Naukowe, Warszawa 1977

[25] Blikle Andrzej, *Toward Mathematical Structured Programming*, Formal Description of Programming Concepts (Proc. IFIP Working Conf. St. Andrews, N.B Canada 1977, E.J Neuhold ed. pp. 183-2012, North Holland, Amsterdam 1978

[26] Blikle Andrzej, *On Correct Program Development*, Proc. 4th Int. Conf. on Software Engineering, 1979 pp. 164-173

[27] Blikle Andrzej, *On the Development of Correct Specified Programs*, IEEE Transactions on Software Engineering, SE-7 1981, pp. 519-527

[28] Blikle Andrzej, *The Clean Termination of Iterative Programs*, Acta Informatica, 16, 1981, pp. 199-217.

[29] Blikle Andrzej, *MetaSoft Primer — Towards a Metalanguage for Applied Denotational Semantics*, Lecture Notes in Computer Science, Springer Verlag 1987

[30] Blikle Andrzej, *Denotational Engineering or from Denotations to Syntax*, red. D. Bjørner, C.B. Jones, M. Mac an Airchinnigh, E.J. Neuhold, *VDM: A Formal Method at Work*, Lecture Notes in Computer Science 252, Springer, Berlin 1987

[31] Blikle Andrzej, *Three-valued Predicates for Software Specification and Validation*, first published in VDM'88, VDM: The Way Ahead, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, pp. 243-266, later republished in Fundamenta Informaticae, January 1991

[32] Blikle Andrzej, *Denotational Engineering*, Science of Computer Programming 12 (1989), North Holland

[33] Blikle Andrzej, *Why Denotational — Remarks on Applied Denotational Semantics*, Fundamenta Informaticae 28, 1996, pp. 55-85

[34] Blikle Andrzej, *An Experiment with a user manual based on denotational semantics*, preprint 2019, DOI: 10.13140/RG.2.2.23355.67366

[35] Blikle Andrzej, *An Experiment with denotational semantics*, SN Computer Science, (2020) 1: 15. https://doi.org/10.1007/s42979-019-0013-0, Springer

[36] Blikle Andrzej, Jarosław Deminet, *Komputerowa edycja dokumentów dla średnio zaawansowanych*, (Computer-assisted edition of documents for medium-advanced authors), Helion 2020

[37] Blikle Andrzej, Mazurkiewicz Antoni, *An algebraic approach to the theory of programs, algorithms, languages and recursiveness*, Proc. International Symposium and Summer School on Mathematical Foundations of Computer Science, Warsaw-Jabłonna, 1972.

[38] Blikle Andrzej in cooperation with Schubert Aleksander, Dziubiak Marian, Kamas Tomasz, *Lingua-WU Report and a diary of the development of its implementation,* a manuscript in statu nascendi

[39] Blikle Andrzej, Tarlecki Andrzej, *Naïve denotational semantics*, Information Processing 83, R.E.A. Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), © IFIP 1983

[40] Blikle Andrzej, Tarlecki Andrzej, Thorup Mikkel, *On conservative extensions of syntax in system development*, Theoretical Computer Science 90 (1991), 209-233

[41] Bordis Tabea, Runge Tobias, Schaefer Ina, *Correctness-by-Construction for Feature-Oriented Software Product Lines*, Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426959

[42] Branquart Paul, Luis Georges, Wodon Pierre, *An Analytical Description of CHILL, the CCITT High-Level Language*, Lecture Notes in Computer Science vol. 128, Springer-Verlag 1982

[43] Chailloux Emmanuel, Manoury Pascal, Pagano Bruno, *Developing Applications With Objective Caml*, Editions O'REILLY, http://www.editions-oreilly.fr

[44] Chomsky Noam, *Three models for the description of language*, IRE Transactions of Information Theory, IT2, 1956

[45] Chomsky Noam, *Syntactic Structures*, Hague 1957

[46] Chomsky Noam, *On certain formal properties of grammars*, Information and Control, 2, 1959

[47] Chomsky Noam, *Context-free grammar and pushdown storage*, MIT Research Laboratory Electrical Quarterly Progress Reports 65, 1962

[48] Cohn P.M., *Universal Algebra*, D. Reidel Publishing Company 1981

[49] Dijkstra Edsger, W., *goto statements considered harmful*, Communications of ACM, 11, 1968, pp. 147-148

[50] Dijkstra Edsger, W., *A constructive approach to the problem of program correctness*, BIT 8 (1968)

[51] Dijkstra Edsger, W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1976

[52] DuBois Paul, *MySQL*, Wydanie II rozszerzone, Mikom, Warszawa 2004

[53] Floyd Richard W., *Assigning meanings to programs*, Appl. Math. Comput. 19, 1967, pp. 19-32

[54] Forta Ben, *SQL w mgnieniu oka*, Helion 2015

[55] Ginsburg Seymur, *The mathematical theory of context-free languages*, New York 1966

[56] Ginsburg Seymur, Rice, H.G., *Two Families of Languages Related to Algol*, Journal of the Association of Computing Machinery, 9 (1962)

[57] Goguen, J.A., *Abstract errors for abstract data types*, in Formal Descriptions of Programming Concepts (Proc. IFIP Working Conference, 1977, E.Neuhold ed.), North-Holland 1978

[58] Goguen, J.A., Thatcher J.W., Wagner E.G., Wright J.B., *Initial algebra semantics, and continuous algebras*, Journal of ACM 24 (1977)

[59] Gordon M.J.C., *The Denotational Description of Programming Languages*, Springer Verlag, Berlin 1979

[60] Gruber Martin, *SQL*, Helion 1996

[61] Hoare C.A.R., *An axiomatic basis for computer programming*, Communications of ACM, 12, 1969, pp. 576-583

[62] Jensen Kathleen, Wirth Niklaus, *Pascal — User Manual and Report*, Springer Verlag 1975

[63] Jones Cliff B., *Understanding Programming Languages*, Springer 2020

[64] Kleene Steven Cole, *Introduction to Metamathematics*, North-Holland 1952; later republished in years 1957, 59, 62, 64, 67, 71

[65] Konikowska Beata, Tarlecki Andrzej, Blikle Andrzej, *A three-valued Logic for Software Specification and Validation*, w tomie VDM'88, VDM: The Way Ahead, Proc. 2nd, VDM-Europe Symposium, Dublin 1988, Lecture Notes of Computer Science, Springer Verlag 1988, pp. 218-242

[66] Landin, P. *The mechanical evaluation of expressions*, BSC Computer Journal, 6 (1964), 308-320

[67] Leszczyłowski Jacek, *A theorem of resolving equations in the space of languages*, Bull. Acad. Polonaise de Science, Série de Sci. Math. Astronom. Phys. 19 (1971)

[68] Leroy Xavier, Doligez Damien, Frisch Alain, Garrigue Jacques, Rémy Didier, Vouillon Jérôme, *The OCaml system release 4.10, Documentation and user's manual*, February 21, 2020, Copyright © 2020 Institut National de Recherche en Informatique et en Automatique

[69] Madey Jan, *Od wnioskowania gramatycznego do walidacji specyfikacji wymagań*, w tomie „Symulacja w badaniach i rozwoju", tom 6, Politechnika Białostocka; na Researchgate https://www.researchgate.net/publication/283225534_Od_wnioskowania_gramatycznego_do_walidacji_specyfikacji_wymagan_From_grammatical_inference_to_validation_of_requirements_specification

[70] Madey J., Matwin S., *Pascal — opis języka*, Sprawozdania IInf UW nr 54 oraz 55, Wydawnictwa Uniwersytetu Warszawskiego, Warszawa 1976

[71] Mazurkiewicz Antoni, *Proving algorithms by tail functions*, Information and Control, 18, 1971, pp. 220-226

[72] Mazurkiewicz Antoni, *Introduction to Trace Theory (tutorial)*, in …???

[73] Mazurkiewicz Antoni, *Compositional Semantics of Pure Place/Transition Systems*, Advances in Petri Nets, Lecture Notes in Computer Science (G. Rozenberg ed.), vol. 340 (1988) pp 307-330

[74] McCarthy John, *A basis for a mathematical theory of computation*, Western Joint Computer Conference, May 1961 later published in Computer Programming and Formal Systems (P. Brawffort and D. Hirschberg eds), North-Holland 1967

[75] Microsoft Press (opr. w. polskiej Piotr Stokłosa), *Microsoft Access 2000 — wersja polska*, Wydawnictwo RM, 2000

[76] Naur Peter (ed.), *Report on the Algorithmic Language ALGOL60*, Communications of the Association for Computing Machinery Vol. 3, No.5, May 1960

[77] Niemiec Andrzej, *Wielkość współczesnego oprogramowania*, Biuletyn PTI nr 4-5, 2014

[78] Norton Peter, Samuel Alex, Aitel David, Eriv Foster-Johnson, Richardson Leonard, Diamond Jason, Parker Aleatha, Michael Roberts, *Python od podstaw*, Wydawnictwo Helion 2006

[79] Parnas D.L., Asmis G.J.K., Madey J., Assessment *of Safety-Critical Software in Nuclear Power Plants*, Nuclear Safety 32, 2, April-June 1991, str. 189-198.

[80] Paszkowski Stefan*, Język ALGOL 60*, PWN 1965

[81] Plotkin Gordon D, *An operational semantics for CSP*, in: Formal Description of Programming Concepts II, D. Bjørner, ed., North-Holland, Amsterdam, pp. 199–225.

[82] Sephens Ryan, Jones D. Arie, Plew Ron, *SQL w 24 godziny*. Helion 2016

[83] Stoy, J.E., Denotational Semantics: *The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA 1977

[84] Scott D., Strachey Ch., *Towards a mathematical semantics of computer languages*, Technical Monograph PRG-6, Oxford University 1971.

[85] Tarski Alfred, *Pojęcie prawdy w językach nauk dedukcyjnych*, Prace Towarzystwa Naukowego Warszawskiego, Nr 34, Wydział III, 1933, str.35

[86] Tucker J. V., Zucker J. I.. *Program Correctness over Abstract Data Types, with Error-State Semantics.* North-Holland and CWI Monographs, Amsterdam, 1988.

[87] Turing Alan, *On checking a large routine*, Report of a Conference on High-Speed Calculating Machines, University Mathematical Laboratory, Cambridge 1949, pp. 67-69.

[88] Vera (del) Pilar Castillo, Curley Martin, Fabry Eva, Gottiz Michael, Hagedorn Peter, Herczog Edit, Higgins John, Joyce Alexa, Korte, Werner, Lanvin Bruno, Parola Andrea, Straub Richard, Tapscott Don, Vassallo John, *Manifest w sprawie e-umiejętności*, European Schoolnet (EUN Partnership AISBL)

[89] Viescas John, *Podręcznik Microsoft Access 2000*, wydawnictwo RM 2000

Blikle Andrzej in cooperation with Schubert Aleksander, Alenkiewicz Joachim, Dziubiak Marian, Kamas Tomasz, *Lingua-WU Report and a diary of the development of its implementation,* a manuscript in statu nascendi

Bordis Tabea, Runge Tobias, Schaefer Ina, *Correctness-by-Construction for Feature-Oriented Software Product Lines*, Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE ’20), November 16–17, 2020, Virtual, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426959

14.2 Index of terms and authors

abortion state 254
abstract error 33
abstract Petri net 264
abstract syntax 41, 135
acceptability relation 87
actual parameter 107
algebra of data 62
algebra of symbolic behaviors 248
algorithmic condition 178
ambiguous algebra 45
ambiguous grammar 48
anchored class transformer 177
Apt K. 148
arity of a function 38
array 63
Asmis G.J.K. 149
assertion 176
assignment instruction 104
atom of a net 264
atomic bundle 252
atomic computation 252
atomic metaprogram 185
atomic net 262
atomic preamble 185
atomic specinstruction 185
Bakker (de) Jaco 197
basic data 208
basic type 208
binary relation 27
bundle of computations 252
carrier of a value 80
carrier of an algebra 38
Cartesian power 18
catalysing condition 182
chain 23
chain-complete partially ordered set 23
child 213
child column 229
child table 229
Chomsky's polynomial 27
clan of a body 66
clan of column type 224
clan of yoke 74
class 83
class transformer 126
clean termination 155
clean total correctness 154, 254
CLI 208
cocoon 268
codomain of a relation 28
co-hereditary condition 183
Collatz hypothesis 156
colloquial syntax 52, 59, 142
column marking 224
column orphan 231
column table-content 226
column type 224
column-type expressions 238
column-yoke expression 237
composition of bundles 252
composition of computations 251
composition of nets 263
compositionality 54
computable partiality of functions 34
computation 251
concatenation of languages 25
concatenation of tuples 22
concrete semantics 58
concrete syntax 58, 138
condition 170
conservative extension of an algebra 39
conservative transfer 73
constant 84
constant of an algebra 38
constructor of an algebra 38
consumed condition 182
content-subordination relation 228
context-free algebra 47
context-free grammar 25
context-free language 25
continuation 55, 151
continuous function 23
converse relation 28
copy rule 55
covering relation 87
cursor 219
cursor declaration 219
cursor grasp 219
dangling reference 78
data 63
database 228
declaration of class 125
declaration of variable 124
declaration section 109
deep attribute 80
denotation 54
dependency relation 256
deposit 78
diligent transfer constructor 73
divisors of zero 251

domain 31
domain of a function 19
domain of a relation 28
eager evaluation 35
empty class....84
empty data....222
equational grammar 26
equationally definable language 27
error transparent condition....171
error trap 105
error-handling mechanism 105
essential condition 182
execution of a net....262
existential quantifier 17
extension of a signature 39
extension of an algebra 39
external name of a class 84
Fermat theorem....156
field 210
five-step method 59
fixed point equation 24
flow relation....261
flow-diagram....150
Floyd Richard 148
foreign key....213
formal language 25
formal-parameter 109
full dependency....256
function....28
functional method 107
general quantifier 17
Goguen Joe 14
goto instruction 54
halting property....155
handling regime 86
hereditary component of a state 221
hereditary condition 183
Hoare C.A.R 148
holistic column-yoke 223
homomorphism (many-sorted) 39
hosting state 89
identity function....20
identity relation....27
immanent condition 184
immunizing condition....184
imperative method 107
independency relation....256
indicator of an entity 86
induced condition....183
infinitistic bundle 252
initial execution of a net 262
inner object 80
instruction 103
integrity constraints 230
invariant of a loop 191
item 86
iteration of a function....19
iterative program....150
Jaco de Bakker paradox 197
joint predicate 216
jump instruction 150
kernel of a homomorphism 40
Kleene's propositional calculus....36
labeled row....225
lazy evaluation 35
least element 22
least fixed point of a function 24
least upper bound 23
left-algorithmic conditions....178
left-hand-side linear equation 151
limit of a chain 23
Lingua 61
Lingua-MV 170
Lingua-V....170
list 63
list view of an objecton....80
LL(k) grammar 52
Madey J....149
many-sorted language 26
mapping 18
Mazurkiewicz A....148
McCarthy's propositional calculus 35
metacomponent of specprogram....180
metaconditions 179
metadeclaration....180
metainstruction 180
metapredicate....179
MetaSoft 13
method 107
method environment 83
monoid 250
monotone function 23
net 250
new states....232
object....78
object constructor....120
object type....78
objecton....78
of-zones....177
Olderog H.R....148
one-one function 28
on-zones....177
operational semantics....54
origin tag....78
origin tag of store/state 85
orphan reference 79
overwriting of a function 21
parent 213

parent column 229
Parnas D.L. 149
partial correctness 154, 254
partial function 18
partial order 22
partial precondition 155
partially ordered set 22
path of transitions 262
pattern 169
perpetual condition 183
Petri net 261
polynomial 27
polynomial equation 153
power of a language 25
preamble of a program 95
prefix of a computation 253
pre-procedure 108
primary condition 183
primary constructor 64
primary key 213
prime data 62
prime-data constructors 62
principle of simplicity 57
private visibility 88
procedure 107
procedure indicator 132
procedure signature 107
procedure signatures 84
pseudotype 84
quantified column-yoke 223
quasinet 250
query 216, 246
reachability function 262
reachable algebra 43
reachable subalgebra 43
record 63
record attribute 63
reference 78
reference carried by objecton 79
reference expression 103
reference parameter 107
reflexive domain 55, 108
reflexivity 22
register 205
register-expression 205
register-identifier 205
register-invariant 205
relation 27
replicated denotation 233
rescue action 105
resilient condition 182
restoring transformation 59, 135
restriction of a signature 39
right-algorithmic conditions 178
roll-back value 214
row table-content 226
row-yoke expression 238
Scott D. 151
semantics 54, 143
semantics of abstract syntax 43
sequential behavior of a net 262
sequential composition of relations 28
signature of an algebra 38
signature of constructor 38
similar algebras 39
similar signatures 40
simple recursion 153
skeleton function 46
skeleton homomorphism 49
skeleton of a function 47
sort of a function 38
specified programs 177
SQL 208
state 84
store 84
Strachey Ch. 151
strong composition of a bundle 254
strongest partial postcondition 180
strongly prefixed grammar 52
structural constructor 151
structure view of an objecton 80
structured induction 54
structured instruction 104
structured programming 151
subalgebra 39
subordination relation 212, 229
surface attribute 80
symbolic behavior of a net 263
synchronization of languages 260
synchronization of trace languages 260
syntactic algebra 47
syntax 54
table 210, 227
table header 226
table orphan 231
table type 226
table-header expression 239
table-type expressions 239
tail function 151
temporal quantifier 256
token 78
total function 18
total order 22
trace 257
trace equivalence 257
trace language 257
trace projection 259
transaction 214, 246

transition dictionary 267
transition function 261
transitivity 22
truncation of a function 19
trust test 64
truth domain of a condition 171
tuple 21
Turing Alan 148
type record 66
type environment 83
type expression 101
typing discipline 61
unambiguous algebra 45
unambiguous grammar 48
unambiguous key 213
underivablr condition 184
update of a function 21
upper bound 23
usability regime 86
validating programming 168
value 77
value expression 96
value parameter 107
variable 84, 85
view 218, 247
view declaration 218
virtual table 218
visibility categories 88
visibility regime 86
Wagner Eric 14
weak antisymmetricity 22
weak total correctness 154, 254
weak total postcondition 155
weak total precondition 155
weakest total precondition 180
well-formed class 85
well-formed objecton 85
well-formed state 85
while loop 105
word 25
word language 257
Wright Jessie 14
wrt 24
yoke 73
yoke expression 73, 136
zone assertion 178

14.3 Index of notations

() : empty word	{} : empty set/relation	[A] : subset of identity rel.
⊆ : a subset of	⊑ : partial order	♦ : overwriting a function
→ : partial functions	Θ : pseudotype	@ : algorithmic formula
↦ : total functions	{a.i | i=1;n} : a set	■ : end of theorem/proof
⇒ : mappings	(a.i | i=1;n) : a sequence	⇨ : stronger than
● : composition of relations	[a.i/b.i | i=1;n] : a mapping	
© : concatenation	Rel.(A,B) : set of relations	
∃ : there exists		
∀ : for all		