Systems engineering : a formal approach. Part V. Specification language

Eindhoven University of Technology

Department of Mathematics and Computing Science

Systems Engineering: a Formal Approach

Part V : Specification Language

by

K.M. van Hee

93/13

Computing Science Note 93/13
Eindhoven, April 1993

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Information Systems Engineering: a Formal Approach

by

K.M. Van Hee

March 30, 1993

This report is part of a preliminary version of a book that will be published.

Contents

Part V

Specification Language

Chapter 20

Introduction

We specify systems by defining simplex and complex classes on the one hand, and actors on the other hand. For simplex and complex classes we already introduced a graphical language to define important parts, however there is no language to define the values and value types for attributes. Similarly we have introduced a graphical language to define the net model of an actor however we do not have a language to define the processor relations. The specification language will be used to fill these gaps.

expressive comfort

We want to have a language of high *expressive comfort* which means that it is relatively easy to express the concepts we need in a formal way. Since we have chosen to use mathematical concepts to model systems and since we aim to develop precise, well-defined models of systems, it seems natural to choose the *language of mathematics* for our purpose. However there is not such a thing. To give a formal definition of such a language is very difficult and may be impossible. Therefore we define a subset of the mathematical language formally. The main restrictions we make are that we do not allow function-valued functions and that all sets are finite. (As a consequence we do not have for example the limit concept.)

executable model

In modeling systems we often want our models to be *executable*, which means that a computer can simulate the behavior of a system, given a model of it. So the language should at least have an executable subset.

static type system

Another desired feature of the specification language is a *static type system*. This means that all expressions we define in the language have a *type* and that it is possible to verify the assigned types without evaluation of expressions. (So the types can be verified at *compile* time instead of *run* time.) Many errors can be detected in an early stage by type checking.

expressive power

At first sight one may think that *imperative programming* languages like Pascal or C could satisfy our need. They certainly have enough *expressive power*, which means that every concept we possibly need, can be expressed in such a language. (In particular every computable function can be expressed in such a language.) However the *expressive*

comfort is too low and there is no powerful type system. A typical weakness of imperative languages is the *assignment statement*, which enables one variable to have two different values in the same expression. Therefore the semantics and verification of expressions is difficult.

Logical languages like Prolog and functional languages like ML are better candidates, specifically if they have a type system (like ML). However they still do not have the expressive comfort of the usual mathematical notations yet.

Algebraic specification languages like Act One and Obj are also good candidates, while *model-oriented specification* languages like *Z* and VDM are more suitable for our purpose because they allow explicit modeling of data structures.

Our specification language is a subset of *Z* and we give *type rules* and *semantics*. (The name of the language Z is a mark of honor to Zermelo, who was one of the pioneers of the axiomatic foundation of set theory.) A subset of our language is *constructive* and is in fact a *typed lambda calculus*. Specifications in the constructive subset are called *executable specifications*. Our language is *extendible* because we do not limit the number of primitive types and primitive functions. So a systems engineer may add his own primitives.

executable specifications
extendible language

The term *specification language* is usually used for languages to specify data structures and functions or relations on them and not for languages to express complete system models. Therefore we use the term in this sense. Together with the graphical languages to define simplex and complex classes and actors, they form a complete *modeling language*.

specification language
modeling language

There are three important approaches to formal foundation of mathematics, based on: *set theory, predicate logic* and *lambda calculus*. In the approach based on lambda calculus every mathematical concept is expressed as a lambda expression. In set theory every concept is considered to be a set. With only the symbol $\emptyset$ for the empty set, comma's and set brackets, one is able to represent all finite sets. For instance, the natural numbers can be represented by: $0 = \emptyset, 1 = \{\emptyset\}, 2 = \{\emptyset, \{\emptyset\}\}$ and an arbitrary natural number is represented by $n = \{0, 1, \ldots, n-1\}$ (where each natural number n should be replaced by its representation).

lambda calculus
set theory

To express functions, one needs pairs of elements. Suppose a and b are mathematical values represented as sets, then we can represent the pair (a, b) as $\{\{a\}, \{a, b\}\}$. (Note that it is possible to deduce which element is the first and which one is the second element.)

We use a mixture of the three approaches, which may be called a *typed set theory*. Instead of using only sets we use other mathematical constructs such as rows, sequences and tuples, as well. Our construction starts with an arbitrary (but finite) number of primitive sets (called *basic types*) and a fixed set of *constructors*. Everything mathematical concept that can be constructed in this way is called a *finite mathematical value*, or shortly *value*. (In fact all values can be represented as finite sets as we have seen above.) Note that simplexes and complexes are also values, in this sense! Besides finite mathematical values we have *types* which are (finite or infinite) sets of these values and *functions* that map values

typed set theory
finite mathematical value

of one type to values of another type. Note that functions are no values!

The semantics of our language will be expressed in untyped set theory and predicate logic. Because this *meta language* and the specification language are very close, we use the same symbols for semantics and syntax. From the context it will be clear which one we mean. In chapter 21 we consider the mathematical notions we need. In chapter 22 we treat the syntax of the *constructive* part of the language and its mapping to the semantic notions. Here we define types, values and functions that can be executed. In chapter 23 we define the *declarative* part of the language where we introduce predicates, function declarations, schemas and finally scripts. In chapter 24 we consider some methods for construction of functions. In chapter 25 we give some methods for the definition of complex classes and processor relations.

Chapter 21

Semantic concepts

In this chapter we introduce the *semantic concepts* of the language, in particular the notions of *values*, *types* and *functions*. (So the only syntax in this chapter belongs to the meta language.) We start with values and types.

21.1 Values and types

The mathematical concepts we will consider, have to belong to specific sets called *types*. The elements of types are called *finite mathematical values* or simply: *values*. We postulate the existence of *basic types*.

Definition 21.1 A *basic type* is a finite or countable set. The elements of these sets are called *constants*. The set of basic types $\mathcal{B}$ is finite and contains at least:

basic type
constants

- $\emptyset$, the *empty* type,
- $\mathbb{B}$ the type of *truth values*, i.e. $\mathbb{B} = \{true, false\}$,
- $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$, the types of *natural*, *integer* and *rational* numbers.

The basic types are mutually disjoint. The set of all constants is denoted by C, i.e. $C = \bigcup \mathcal{B}$.
□

At first sight it might look strange that we assume $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$ mutually disjoint. There is a good reason to make this assumption because we can construct $\mathbb{Z}$ from $\mathbb{N}$ and $\mathbb{Q}$ from $\mathbb{Z}$. In these constructions an integer is an equivalence class of pairs of naturals and a rational is an equivalence class of pairs of integers: for example -5 is the equivalence class of all pairs of naturals (x, y) such that $x + 5 = y$. So according to this construction the types are really disjoint. However there are isomorphisms that embed $\mathbb{N}$ and $\mathbb{Z}$ in $\mathbb{Q}$. The values in $\mathbb{N}$ are denoted by $\{0, 1, 2, \ldots\}$, the values in $\mathbb{Z}$ as in $\mathbb{N}$ but with a sign (for example +3 and -3) and the values in $\mathbb{Q}$ as pairs separated by / and with a sign (for example +3/4). This notation will be the same in the specification language in the next chapter.

At first sight the $\emptyset$ as type seems to be useless because it does not contain values. There is however a type constructor ($\mathbb{F}$), that can build on $\emptyset$ to obtain types that contain values.

Next we introduce *value constructors* to construct new values out of constants and one special value $\perp$ that does not belong to the basic types. It will have a particular meaning "unknown" or "non-existent". The value $\perp$ is used in the following cases:

- to express that a processor does not consume a token from a certain input connector,
- to express that a function is not defined for a certain argument,
- in three-valued logic, where there is besides *true* and *false* a third value *unknown*.

Definition 21.2 All constants and $\perp$ are values.
We consider four *value constructors*:

set constructor

1. *Set constructor*:
   If, for $n \geq 0$, $b_1, \ldots\ldots, b_n$ are values, then

$$\{b_1, \ldots, b_n\}$$

   is also a value called a *set* although it is always a finite set. The set constructor forms finite sets of values. In particular $\{\}$ is a value, called the *empty set*.

empty set

row constructor

2. *Row constructor*:
   If, for $n \geq 0$, $b_1, \ldots\ldots, b_n$ are values, then

$$(b_1, \ldots, b_n)$$

   is also a value, called a *row*. A row of two elements is called a *pair* and $()$ is called the *empty row*.

pair
empty row

sequence constructor

3. *Sequence constructor*:
   If, for $n \geq 0$, $b_1 \ldots, b_n$ are values, then

$$\langle b_1, \ldots, b_n \rangle$$

   is a value called a *sequence*. In particular $\langle\,\rangle$ is a value, called the *empty sequence*.

sequence
empty sequence

tuple constructor

4. *Tuple constructor*:
   Tuples are constructed with the use of *attributes*. Therefore we postulate the existence of a countable set of *attributes* L, which is disjoint with all basic types and does not contain $\perp$.
   If, for $n \geq 0$, $b_1, \ldots, b_n$ are values and $\ell_1, \ldots, \ell_n$ are different *attributes* then

$$\{\ell_1 \mapsto b_1, \ldots, \ell_n \mapsto b_n\}$$

   is a value, called a *tuple*. The elements $(l \mapsto b)$ of a tuple are called *components*. In particular $\{\}$ denotes the *empty tuple* (which is the same as the empty set).

tuple
components
empty tuple

□

Note that $\emptyset$ and $\{\}$ are different concepts: the first one is a type while the second one is a value.

A sequence only differs, at this moment, from a row by the kind of brackets. Later, when we introduce types, we will see the "real" difference between rows and sequences. A tuple is in fact a set of pairs; the first component of a pair is an attribute. Since the attributes are unique it is even a function. We use a different notation for the pairs in a tuple to distinguish them from "normal" pairs, because the first element of these pairs is an attribute instead of a value.

The set of all possible values, that can be formed from the constants and $\perp$ by finite application of the value constructors, is called the *free value universe*. For example

$$(3, \perp, \perp, \langle 4, 5, 6 \rangle)$$

and

$$\{a \mapsto 3, b \mapsto \perp, c \mapsto (4, 5, 6)\}$$

are values. The equality function, denoted by $=$, will have the property that two sets with the same elements but with a different representation are equal. (Note that we consider $=$ to be a Boolean valued function with two arguments, represented in infix notation.) For instance $\{1, 2, 3, 3\} = \{3, 2, 1\}$ is *true*. (Note that $\{1, 2, 3, 3\}$ is a well-formed set here.) Similarly two tuples are equal if they have the same components, for instance

$$\{a \mapsto 2, b \mapsto 3\} = \{b \mapsto 3, a \mapsto 2\} \text{ is true.}$$

However two rows are equal if and only if they have identical representations, so

$$(1, 2, 3, 3) = (3, 2, 1) \text{ is false.}$$

Tuples are important for expressive comfort; their components can be retrieved by specifying a attribute, while we have to compute the position of an element in a row in order to retrieve it.

Definition 21.3 Let $\mathcal{B}$ be a non-empty finite set of basic types. The set FU, called the *free value universe*, is the smallest set such that:

free value universe

1. $\perp \in FU$,
2. $C \subset FU$,
3. FU is closed under the four value constructors.

□

This set contains too many values. For instance the value $\{3, \mathit{true}, (1, \mathit{false})\}$ is not a value we want to consider, for a set to be a value should contain only values of the same type. Analogously to value constructors we define *type constructors*. Remember, a type is just a set of values.

Definition 21.4 The four *type constructors* are:

set type constructor

1. *Set type constructor*:
   If T is a type then
   $$\mathbb{F}(T)$$
   set type
   denotes the type of all finite sets of values of T and is called a *set type*.

product type constructor

2. *Product type constructor*:
   If $T_1, \ldots, T_n$, for $n \geq 0$, are types then
   $$T_1 \times \ldots \times T_n$$
   denotes the type of all rows $(t_1, \ldots, t_n)$ where $t_i \in T_i$ for $i \in \{1, \ldots, n\}$, and is called a *product type*.
   product type

sequence type constructor

3. *Sequence type constructor*:
   If T is a type then
   $$T^*$$
   is the type of all finite sequences of values of type T.

tuple type constructor

4. *Tuple type constructor*:
   If $T_1, \ldots, T_n$ are types and $\ell_1, \ldots, \ell_n$ are distinct attributes, then
   $$[\ell_1 : T_1, \ldots, \ell_n : T_n]$$
   is the type of all tuples $\{\ell_1 \mapsto t_1, \ldots, \ell_n \mapsto t_n\}$ where $t_i \in T_i$ for $i \in \{1, \ldots, n\}$.

□

Two values of type T^* may therefore have different length but their elements will be all of the same type.

Note that two tuple types are equal if they have the same set of attributes and for each attribute the same type. Also note that

$$A \times (B \times C) \neq A \times B \times C.$$

Next we introduce the *type universe* TU and the universe of allowed values U. (Note that the notion of a tuple in the specification language differs from the tuple we used to define frameworks!)

Definition 21.5 Let $\mathcal{B}$ be a non-empty, finite set of basic types. The set TU is the smallest set such that:
type universe

- $\forall B \in \mathcal{B} : B_\perp \in TU$,
  where $B_\perp = B \cup \{\perp\}$,
- TU is closed under the four type constructors.

The *value universe* U satisfies: **value universe**

$$U = \{x \mid \exists T \in TU : x \in T\}$$

□

Note that it is not excluded that a value from U belongs to more than one type.

Theorem 21.1 All elements of the value universe are in the free value universe, i.e. $U \subset FU$.

Proof. First note that (by definition) all constants and $\perp$ belong to U. If $x \in U$ then there is a type $\tilde{T}$ such that $x \in \tilde{T}$. We use structural induction to show $\tilde{T} \subset FU$. Let $T, T_1, \ldots, T_n$ belong to TU and let them be subsets of FU. Assume $\tilde{T} = I\!F(T)$. Then $\tilde{T} \subset FU$ since all elements of $\tilde{T}$ are finite sets of elements of FU. Next assume $\tilde{T} = T_1 \times ... \times T_n$. Then it is also a subset of FU because all elements of $\tilde{T}$ are rows of elements of FU. The other cases are similar.
□

Theorem 21.2 All types in the type universe are countable.

Proof. We use structural induction. First note that, by definition all basic types, extended with $\perp$, are countable.

- Assume that type T is countable. So T is isomorphic with $I\!N$.

  Consider $I\!F(T)$. Each element of $I\!F(T)$ can be represented as an infinite sequence of 0's and 1's: its n-th element is a 1 if and only if the n-th element of T (according to the isomorphism) is in the set. Hence every element of $I\!F(T)$ has finitely many 1's and so it represents a natural number in binary notation. So $I\!F(T)$ is also isomorphic to the set of natural numbers, and therefore countable.

- Assume that $T_1, \ldots, T_k$ are countable types. Consider $T_1 \times \ldots \times T_k$. This set is isomorphic with the set $I\!N^k$, because each T_i is isomorphic with $I\!N$. This set is *countable* because for each natural number m the set of rows $(n_1, \ldots, n_k)$ with $n_1 + \ldots + n_k = m$ is finite, and therefore we can count them (first the set with $m = 0$, then $m = 1$ etc.).

- Assume T is countable, so it is isomorphic with $I\!N \backslash \{0\}$. Consider T^*. This set is isomorphic with the set of all finite sequences of natural numbers (unequal to zero). For each $k \in I\!N$ the number of sequences with sum k is finite. Therefore we can count them.

- Assume that $T_1, \ldots, T_k$ are countable types and that $\ell_1, \ldots, \ell_k$ are distinct attributes. Then $[\ell_1 : T_1, \ldots, \ell_k : T_k]$ is isomorphic with $[T_1 \times \ldots \times T_k]$ and therefore it is countable.

So we conclude that all $T \in TU$ are countable.
□

Theorem 21.3 The sets U and FU are countable.

Proof. The proof is an exercise.
□

Next we assume that every basic type has total ordering, denoted by $\leq$. We will see how this ordering induces an ordering for every type.

lexicographical ordering

Definition 21.6 Let $\leq$ be an ordering on every basic type. With induction we extend $\leq$ to every type in TU. This ordering is called the *lexicographical* ordering. (Let $x < y$ be an abbreviation for $x \leq y \wedge x \neq y$.)

- $\forall c \in C : \bot \leq c$,
- If T is already ordered by $\leq$ then we extend $\leq$ to $\mathbb{F}(T)$ as follows: let $\{a_1, \ldots, a_m\}, \{b_1, \ldots, b_n\} \in \mathbb{F}(T)$, such that

  1. $\forall i \in \{1, \ldots, m-1\} : a_i < a_{i+1}$
  2. $\forall i \in \{1, \ldots, n-1\} : b_i < b_{i+1}$

  then

  $$\{a_1, \ldots, a_m\} \leq \{b_1, \ldots, b_n\}$$

  if and only if one of the following conditions holds:

  $$\exists k \in \{1, \ldots, min(m,n)\} : (\forall i \in \{1, \ldots, k-1\} : a_i = b_i) \wedge a_k < b_k,$$

  $$m \leq n \wedge \forall i \in \{1, \ldots, m\} : a_i = b_i.$$

  In case $p, q \in \mathbb{F}(T)$ do not satisfy (1) and (2), we can find equivalent elements $\tilde{p}, \tilde{q} \in \mathbb{F}(T)$, i.e. $\tilde{p} = p$ and $\tilde{q} = q$, such that $\tilde{p}$ and $\tilde{q}$ do satisfy (1) and (2). Then we define $p \leq q$ if and only if $\tilde{p} \leq \tilde{q}$.
- If $T_1, \ldots, T_n$ are already ordered by $\leq$ then we extend $\leq$ to $T_1 \times \ldots \times T_n$ as follows

  let $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in T_1 \times \ldots \times T_n$

  then

  $$(a_1, \ldots, a_n) \leq (b_1, \ldots b_n)$$

  if and only if one of the following conditions holds:
  $\exists k \in \{1, \ldots, n\} : (\forall i \in \{1, \ldots, k-1\} : a_i = b_i) \wedge a_k < b_k$,
  $\forall k \in \{1, \ldots, n\} : a_k = b_k$.

- If T has already been ordered then we extend $\leq$ to T^* like we did for $\mathbb{F}(T)$.
- If $\ell_1, \ldots, \ell_n$ are distinct attributes and $T_1, \ldots, T_n$ are types, already ordered by $\leq$, then we extend $\leq$ to $[\ell_1 : T_1, \ldots, \ell_n : T_n]$ as follows:

  - introduce an ordering (also denoted by $\leq$) on the attribute set L,

- map each tuple to a row in $T_{\ell_{i_1}} \times \ldots \times T_{\ell_{i_n}}$ where $\ell_{i_1} \leq \ldots \leq \ell_{i_n}$, and call this map f,
- two tuples x and y satisfy $x \leq y$ if and only if $f(x) \leq f(y)$. (The last ordering is defined above.)

□

In fact the relation $\leq$ is not an ordering on tuples and sets but on the *equivalent classes* with respect to the equality function of tuples and sets!

Theorem 21.4 For all $T \in TU$, $\leq$ is an ordering relation on T.

Proof. By structural induction the proof is an immediate consequence of definition 21.6.
□

Note that sets can be represented in a *normal form* which is the representation of a set where the elements are arranged in ascending order and duplicates are left out. The normal forms are representatives of equivalence classes.
Similarly we introduce an equivalence relation on tuple types and we use it to define a *normal form* for tuple types.
We also introduce another type constructor called *join*, denoted by $\bowtie$.

normal form of a set

Definition 21.7 Let $[k_1 : S_1, \ldots, k_m : S_m]$ and $[\ell_1 : T_1, \ldots, \ell_n : T_n]$ be two tuple types.
They are called *equivalent* if and only if:

tuple equivalence

- $m = n$,
- there is a permutation $(i_1, \ldots, i_n)$ of $(1, \ldots, n)$ such that $\forall j \in \{1, \ldots, n\} : k_j = l_{i_j} \ \wedge \ S_j = T_{i_j}$.

Two tuples are *equivalent* if and only if they have the same components, may be in different order.
They are called *compatible* if and only if:

tuple compatibility

$$\forall i \in \{1, \ldots m\}, j \in \{1, \ldots, n\} : k_i = \ell_j \Rightarrow S_i = T_j$$

And if they are compatible their *join* is denoted by:

tuple join

$$[k_1 : S_1, \ldots, k_m : S_m] \bowtie [\ell_1 : T_1, \ldots, \ell_n : T_n]$$

and is equal to the set of all tuples

$$\{k_1 \mapsto s_1, \ldots, k_m \mapsto s_m\} \cup \{\ell_1 \mapsto t_1, \ldots, \ell_n \mapsto t_n\}$$

such that

$$\forall i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\} : k_i = \ell_j \Rightarrow s_i = t_j$$

□

We assume there is an ordering on the set of attributes L and we say that a tuple type with attributes in ascending order is in *normal form*. For example, if $k \leq \ell$ then the second tuple type of

$$[\ell : T, k : S] \text{ and } [k : S, \ell : T]$$

is the normal form of the first one.
Similarly we say "a tuple is in normal form" if its attributes are in ascending order.

Note that we have values that do not contain any constants, for example:

$$(\langle\rangle), (\{\}), \{\}, \{\{\}\}, \{\langle\rangle\}, \langle\{\}\rangle, (\{\bot\}, \langle\rangle), (\bot).$$

They are examples of *singular values*.

simple singular value

Definition 21.8 A *simple singular value* is a value that does not contain a constant.

singular value

A *singular value* is a simple singular value or a value that contains a simple singular value.

regular values

All other values are called *regular values*.
□

Singular values may belong to more than one type, for example $\{a \mapsto 3, b \mapsto \{\}\}$ belongs to $[a : \mathbb{N}, b : \mathbb{F}(A)]$ as well as to $[a : \mathbb{N}, b : \mathbb{F}(A \times A)]$, for some type A. However regular values have a unique type.

Theorem 21.5 Regular values have a unique type.

Proof. We will show that two types T_1 and T_2 have only singular values in common, which is equivalent with the assertion of the theorem.

Suppose T_1 and T_2 are different basic types extended by $\bot$. Then $\bot$ is their only common value, because basic types are disjoint by definition. If T_1 and T_2 are types of different structure, i.e. they belong syntactically to different categories (the categories are: set type, product type, sequence type and tuple type), then it is easy to verify that they have no regular values in common. The only difficulty occurs with tuples and sets because they have the same brackets. However the only common value is $\{\}$, which is singular.

Now assume $T_1 = \mathbb{F}(A)$ and $T_2 = \mathbb{F}(B)$ and A and B have no regular value in common. Suppose now that $\{a_1, \ldots, a_n\}$ is a regular value and that it belongs to $T_1 \cap T_2$. Then at least one of the elements $a_1, \ldots, a_n$ is a regular value and this value should belong to A and B. This is a contradiction.

Next consider $T_1 = A_1 \times \ldots \times A_n$ and $T_2 = B_1 \times \ldots \times B_n$, where A_i and B_i have no regular value in common, for all $i \in \{1, \ldots, n\}$. Suppose now T_1 and T_2 have a regular value in common, say $(c_1, \ldots, c_n)$. Then, for at least one i the value c_i is a regular value and should occur in A_i and B_i, which is a contradiction.

The same arguments apply to the other two cases where T_1 and T_2 belong to the same syntactical category.
(The proof of the other cases is an exercise.)
□

Hence there exists a function *type* that assigns to regular values the type it belongs to.

21.2 Functions

We have a universe U of values, however we have not yet defined *functions* on this universe. The set of all total functions UF is called the *function universe*, i.e. **function universe**

$$UF = U \to U.$$

Almost all functions we are interested in, are defined as *partial* functions. For example the function *pick* is only defined for sets and not for sequences. In order to make all functions total, we define them as $\perp$ outside their *meaningful domain*. So we assume all functions to be total on U.

Each function has a *name* and a *graph*. Recall that the graph of a function is a set of pairs, such that the first elements of these pairs are unique. Note that a function may have an infinite graph, so a graph of a function is not a value in general.

We will represent function application with brackets, so f applied to a is represented by $f(a)$. A function of more than one variable, for example a and b, is in fact a function on a row, namely (a,b), which can be considered as one variable. If we apply a function to a row we only use one pair of brackets, so we write $f(a,b)$ instead of the more consequent notation $f((a,b))$.

Although types are countable sets, UF is *uncountable*! To verify this, take the set of functions $\mathbb{N} \to \mathbb{B}$.
Since $\mathbb{B}$ is isomorphic with the set $\{0,1\}$, the set of functions $\mathbb{N} \to \mathbb{B}$ is isomorphic with all real numbers in the interval $[0,1]$, which is an uncountable set. (To understand this note that each element of $\mathbb{N} \to \mathbb{B}$ can be considered as a binary fraction.) However, in specifications we will only use countably many functions: the ones that can be defined in our language, which only has countable many sentences. The functions we will use are *constructed* from a given, countable set of *primitive functions*, using *abstraction* and *recursion* (see chapter 22).

There may be more than one definition for the same graph. Since each definition may have its own name, we may have two functions with different names but with the same graph. Therefore two functions are identical if and only if they have the same name and the same graph. (Note however we will not define equality for functions in the specification language, because equality of function graphs is undecidable.) This fact allows us to use the same name for functions having different graphs. This phenomenon is called *overloading* and is often used **overloading**

in mathematics. For instance the function "+" is used for addition of natural numbers but also for addition of complex numbers and vectors in some vector space.

As we are not interested in all values in the free universe, we are not interested in all functions either. We will use the types to characterize the "interesting" functions. These functions have meaningful domains that are types or unions of types. The notion of a *signature* is important here.

function signature

Definition 21.9 A *signature* of a function in UF is a set of pairs of types, i.e. a signature is an element of $I\!F(TU \times TU)$. For $f \in FU$ we denote the signature by $sign(f)$.
□

We will consider two kinds of functions *monomorphic* and *polymorphic* functions.

monomorphic function

Definition 21.10 A *monomorphic* function $f \in UF$ has a signature that is a singleton $\{(T, S)\}$, such that:

$$\forall u \in T : f(u) \in S \vee f(u) = \bot$$

$$\wedge$$

$$\forall u \in U \backslash T : f(u) = \bot.$$

domain type
range type

The type T is called the *domain type* of f and type S the *range type*.
□

So a monomorphic function has one type as meaningful domain. Polymorphic functions have a meaningful domain that consists of several types.

polymorphic function

Definition 21.11 A *polymorphic* function f has a signature $sign(f)$ with at least two elements, such that $\forall u \in U$:

- $(\forall (T \times S) \in sign(f) : u \notin T) \Rightarrow f(u) = \bot$,
- $\exists (T \times S) \in sign(f) : u \in T \ \wedge \ (f(u) \in S \vee f(u) = \bot)$,
- $\forall (T_1 \times S_1), (T_2 \times S_2) \in sign(f)$:

$$u \in T_1 \cap T_2 \Rightarrow f(u) \in S_1 \times S_2 \vee f(u) = \bot.$$

□

The (meaningful) domain of a polymorphic function f is

$$\bigcup \{\pi_1(x) \mid x \in sign(f)\}$$

and the range is

$$\bigcup \{\pi_2(x) \mid x \in sign(f)\}.$$

An example of a polymorphic function is *union*, which assigns to two sets of values of the same type a set of values of that type. Its signature is

$$\{(\mathbb{F}(x) \times \mathbb{F}(x), \mathbb{F}(x)) \mid x \in TU\}.$$

Note that an "overloaded" function name can be considered as the name of one polymorphic function. However in the specification language we make a distinction between overloaded function names and polymorphic functions: an overloaded name has several different signatures assigned to it, while a polymorphic function name has only one signature with type variables in it.

Next we make another distinction between functions: strict and non-strict functions. A *strict* function evaluates to $\perp$ if it is applied to a singular value. A function that is not strict is called *non-strict*. An example of a strict function is $+$, so $3 + \perp$ equals $\perp$. An example of a non-strict function is the *selection* function discussed below. An important subset of the non-strict functions are the *lazy functions*. A function with more than one argument (i.e. a function on rows), is called *lazy* if its function value is already determined if some of the variables are bound by values. If for example

strict function

non-strict function

lazy function

$$\forall x, y : f(3, x) = f(3, y)$$

then the function value is determined by the first argument and then we do not have to evaluate the second one. In particular $f(3, \perp)$ will have the same value. Lazy functions are important in recursive definitions because there we are not able to evaluate all the arguments. The selection function is the most important lazy function.

In our specification language we have some primitive functions, i.e. functions that are not defined in the language but from which we only know the name, the signature and the graph. The graph is only known in implicit form, which means that we assume there is a "retrieval mechanism" that delivers the function value if we give the argument. Primitive functions are therefore defined in the meta language. They are defined informally in part I and formally in the Toolkit. Only two primitive functions are defined here, because they are very important: the *equality* function $(=)$ and the *selection* function *if.then.else.fi*. We use them in infix notation. The function $=$ compares two values and if they are (syntactically) identical or if they are equivalent (in case of sets and tuples) then the function value is *true* else it is *false*. The function is lazy:

$$\perp = \perp \text{ is true, } x = \perp \text{ is false, } \perp = x \text{ is false,}$$

for any $x \neq \perp$.
An application of the selection function reads:

$$\textit{if } a \textit{ then } b \textit{ else } c \textit{ fi}.$$

If a is *true* then the function value is equal to the value of b else to the value of c. The function is lazy: if a is *true* then c may be $\perp$ and if a is *false* then b may be $\perp$. If a is $\perp$ then the function evaluates to $\perp$.

There are in principle countably many primitive functions because of the *projection functions* (π_i, $\Pi_{i,j,\ldots}$). For each attribute (set) there is another projection function. We have chosen for this approach instead of the introduction of *attribute (set) variables*, which would require another kind of lambda constructions in which there are functions with two kinds of variables: value variables and attribute variables. In tools it is easy to generate the projection functions we need.

New functions are defined from primitive functions and already defined functions by means of syntactical constructs that are defined in the next chapter.

Chapter 22

Constructive part of the language

In this chapter we give the syntax to define types, values and functions and we map the syntactical constructs to the semantic notions. This part of the language is the *constructive* or *executable* subset of the language. In the next chapter we introduce the non-constructive part. As noted before, our syntax will be close to the syntax of the meta language. We start with the *meta syntax*, i.e. the syntax we use to define the syntax of the specification language. It is a form of extended BNF.

Definition 22.1 The *meta syntax* follows the next rules: **meta syntax**

- The definition sign for non-terminals is ::=.
- Any part of a syntax in underlined typeface is to be taken literally.
- Any part between '{}' braces may be repeated. So 'a ::= $\{b\}$' is shorthand for 'a := $b \mid b\,a$'.
- Any part between "[ ]" square brackets may be omitted. So 'a ::= $[b]c$' is shorthand for 'a ::= $bc \mid c$'.
- Any part between '<>' triangular brackets may be repeated; each repetition must be preceded by a comma ','. So '$< a >$' is shorthand for '$a[\{,a\}]$'.
- The syntax for identifiers, digits and characters is not further elaborated.

□

We define in fact a *family* of languages. Each language is characterized by a so-called *syntax base* and the syntax rules. (Some components of the syntax base are needed in the subsequent chapters only.)

Definition 22.2 A *syntax base* is a 8-tuple of sets of names **syntax base**

$$(L, C, TV, V, VN, FN, TN, SN)$$

where:

- L is the set of attributes,
- C is the set of constants,
- TV is the set of type variables,
- V is the set of value variables,
- VN is the set of value names,
- FN is the set of function names,
- TN is the set of type names,
- SN is the set of schema names.

The set TN contains the names of the basic types, i.e. at least $\emptyset$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{B}$.
The sets TV and TN are disjoint and $TV = \{\$, \$_1, \$_2, \ldots\}$.
The sets V and VN are disjoint.
The sets L and V satisfy $L \cap V \neq \emptyset$.
□

We assume in the rest of this part that a syntax base is given.

The mapping of expressions to their semantics is performed by an *evaluation function* ϵ. So ϵ maps expressions to values, types or functions. Not all expressions have semantics: only expressions without *free variables* (this notion will be explained later). We will distinguish the expressions in the specification language from expressions in the meta language, by using **[]** brackets for functions that have expressions as domain, for example $\epsilon[\![E]\!]$. Type expressions have the following syntax.

type expression syntax
type definition syntax

Definition 22.3 The syntax of *type expressions* and *type definitions* is:

- *type expression* ::= *type name* | *type variable* | *set type* | *product type* | *sequence type* | *tuple type* | (*type expression*)
- *attribute* $\in L$
- *type name* $\in TN$
- *type variable* $\in TV$
- *set type* ::= $\mathbb{F}$ (*type expression*)
- *product type* ::= *type expression* $\times$ *product list*
- *product list* ::= *type expression* | *product type*
- *sequence type* ::= *type expression**
- *tuple type* ::= *type variable* | [⟨*attribute* : *type expression*⟩] | *tuple type* $\bowtie$ *tuple type*

- *type definition* ::= *type name* $:\equiv$ *type expression*

The set of all type expressions without type variable is denoted by TE.
□

In a subexpression of the form $\$_1 \bowtie \$_2$ of a type expression we know that $\$_1$ and $\$_2$ have to be replaced by *compatible* tuple types only.

The semantics of type expressions is straightforward, since the meta-language and the specification language are so close. For the semantics, i.e. the range of ϵ we use the same symbols as for the specification language. The semantics of type expressions and type definitions is defined formally in the next definition. Note that "type" is a semantic notion and "type expression" a syntactic notion. In expressions of the form "$\epsilon[\![\mathbb{N}]\!] = \mathbb{N}$" the symbol $\mathbb{N}$ is just a name in the left-hand side and the set of natural numbers in the right-hand side.

Definition 22.4 The *evaluation function* ϵ applied to type expressions without type variables, satisfies the following rules. Let all capitals denote type expressions without type variables and with known evaluations. **evaluation function**

1. if T is a basic type then $\epsilon[\![T]\!] = T$,
2. $\epsilon[\![\mathbb{F}(T)]\!] = \mathbb{F}(\epsilon[\![T]\!])$,
3. $\epsilon[\![T_1 \times \ldots \times T_n]\!] = \epsilon[\![T_1]\!] \times \ldots \times \epsilon[\![T_n]\!]$,
4. $\epsilon[\![T^*]\!] = \epsilon[\![T]\!]^*$,
5. $\epsilon[\![[l_1 : T_1, \ldots, l_n : T_n]]\!] = [l_1 : \epsilon[\![T_1]\!], \ldots, \epsilon[\![l_n : T_n]\!]]$,
6. $\epsilon[\![[l_1 : T_1, \ldots, l_n : T_n] \bowtie [k_1 : S_1, \ldots, k_m : S_m]]\!] =$

$$[a_1 : \epsilon[\![P_1]\!], \ldots, a_r : \epsilon[\![P_r]\!]]$$

   if and only if:

   - $\forall i, j : l_i = k_j \Rightarrow \epsilon[\![T_i]\!] = \epsilon[\![S_j]\!]$,
   - $\{a_1, \ldots, a_n\} = \{l_1, \ldots, l_n\} \cup \{k_1, \ldots, k_m\}$,
   - if $a_i = l_j$ then $P_i = T_j$ and if $a_i = k_j$ then $P_i = S_j$,

7. the semantics of a defined type name is equal to the semantics of the defining expression.

□

Note that TE is the syntactical equivalent of the type universe TU. Next we consider the definition of values. We define them by means of *terms* or *value expressions*. The value constructors, defined in the meta language definition 21.2 are already in syntactical form; we will

us them here too. With value constructors we are able to define values by constructing them *explicitly*, which means writing down their representation. However we often need to define values *implicitly* by some expression that indicates how the value should be constructed. For instance, if we want to define the set of all prime numbers that are elements of some given (finite) set s of type $\mathbb{F}(\mathbb{N})$ we do not want to construct this set explicitly. This would require us to check for all elements of s if they are prime or not, but we want to define this set by an expression of the form:

$$\{x : s \mid prime\,(x)\}$$

where *prime* is a Boolean valued function that attains the value *true* if and only if its argument is a prime number. (How we construct or specify these functions will be explained later.) We call such an expression a *set term*. Here we assume we have such a collection of functions, given by their names, for example primitive functions. Note that s in the expression is a *set value* and not a type! Therefore we are sure that the expression denotes a finite value and not an infinite set, such as:

$$\{x : \mathbb{N} \mid prime\,(x)\}.$$

Another example of the use of this set constructor is:

$$\{x : prod\,(s,t) \mid \pi_2(x) = \pi_1(x) \times \pi_1(x)\}$$

which denotes in meta language:

$$\{(y,z) \mid y \in s \wedge z \in t \wedge z = y^2\}.$$

Here s and t are of type $\mathbb{F}(\mathbb{N})$ and *prod* is a function that forms the Cartesian product from two finite sets and $\times$ denotes multiplication. (These functions can be constructed from the primitive functions.) The set constructed above represents in fact a finite function with a domain that is a subset of s and a range that is a subset of t. Of course it is a value itself with type $\mathbb{F}(\mathbb{N} \times \mathbb{N})$. Finite functions or *maps*, are very useful in specifications and therefore it is important to have a convenient notation. The construction above is cumbersome, because we have to define explicitly the set that should contain the range of the map. Therefore we will introduce another syntactical construct to express maps. The same map is expressed, using this construct by $(s \in \mathbb{F}(\mathbb{N}))$:

$$(x : s \mid x \times x).$$

map term
typed lambda expression

This construct is called a *map term* and is formally a *typed lambda expression*.

Now we have seen two constructs to define values implicitly: set term and map term. There is a third one: *function application*. We have seen that already in the examples above: *prime*(3) evaluates to the Boolean value *true* and 4×5 to the number 20. There is also a fourth way by means of the value constructors for terms (instead of constants and $\perp$).

Definition 22.5 The syntax of *terms* is: **term syntax**

- *term* ::= *value variable* | *value name* | *constant* | (*term*) | *value construction* | *application* | *set term* | *map term*
- *value variable* $\in V$
- *value name* $\in VN$
- *constant* $\in C \cup \{\bot\}$
- *function name* $\in FN$
- *value construction* ::= *set construction* | *row construction* | *sequence construction* | *tuple construction*
- *set construction* ::= $\{\langle term \rangle\}$ | $\{\}$
- *row construction* ::= $(\langle term \rangle)$ | $()$
- *sequence construction* ::= $\langle \langle term \rangle \rangle$ | $\langle\rangle$
- *tuple construction* ::= $\{\langle attribute \mapsto term \rangle\}$ | $\{\}$
- *application* ::= *function name* (*term*)
- *set term* ::= $\{$ *value variable* : *domain* | *term* $\}$
- *map term* ::= (*value variable*: *domain* | *term*)
- *domain* ::= *set construction* | *set term*
- *value definition* ::= *value name* := *term* : *type expression*

The type expression in the value definition may not contain type variables.
□

Note that there is for each *value* an identical *value construction* in the language. Therefore all values are terms! A value definition gives a name to a term, such that the name can be used as abbreviation for the term. Not all terms generated by the syntax are allowed. First of all the terms used should be *well-typed*. Typing of terms is defined using a *typing function* τ defined below. For instance, in a set term the term on the right-hand side of the bar should be of the Boolean type, as *prime* in our example above.

In order to be able to define the *typing function* τ and the *evaluation* of terms, i.e. ϵ for terms, we need to the notion of *scope*.

Definition 22.6 In set terms and map terms the term on the right-hand side of the bar is called the *scope*. An occurrence of a variable in a set or map term is said to be *bound*, if it appears left of the colon, or if it appears in the term right of the bar and is the same as a variable occurring left of the colon. **scope** **bounded occurrence**

In case set or map terms are nested, a bound variable on the right-hand side of the bar is bound by the first set or map term with this variable on the left-hand side of the colon, encountered if we go from this variable to the left. The occurrence of a non-bound variable is called a *free variable* occurrence.

free variable occurrence

□

Consider for example the set term

$$\{x : \{1,2,3,4,5,6,\} \mid x \in rng((x : \{1,2,3\} \mid x \times x))\}.$$

The first and second x are bound by the set term, the third, fourth and fifth x by the map term. (Here $\in$ and *rng* are defined functions, they can be found in the toolkit (see appendix C). We only consider semantics of terms without free variables. Hence terms with free variables have semantics with respect to some *context* in which these free variables are bound.

The function τ assigns to terms a type and ϵ a value, only in case the term does not have free variables. We will assume that all free variables in a term differ from bound variables occurring in the term. This can be accomplished by a proper renaming of bound variables. This process is sometimes called *standardizing* and terms that are standardized are called *standard terms*.

standardizing
standard term

We use *substitution* of variables by values. This is denoted by sub- and superscripts. For example

$$b_e^x$$

is the term derived from the term b by *substituting* each free occurrence of x with the term e. In case we have a simultaneous substitution of variables $x_1, \ldots, x_n$ by terms $e_1, \ldots, e_n$ we write

$$b_{(e_1,\ldots,e_n)}^{(x_1,\ldots,x_n)}.$$

type function

Definition 22.7 The *type function* τ assigns to *well-typed* terms a type according to the following rules.
Regular values are well-typed. If t is a regular value then

$$\tau[\![t]\!] = type[\![t]\!],$$

where *type* is the function that assigns to constants their type (cf.theorem 21.5).
Let $t, t_1, \ldots, t_n$ be a well-typed terms.

1. consider a value name n, that is defined in an expression of the form $:= t : T$, then $\tau[\![n]\!] = \epsilon[\![T]\!]$,
2. the type of a constant is the unique basic type to which it belongs,
3. $\tau[\![(t)]\!] = \tau[\![t]\!]$,

4. $\tau[\![\{t_1,\ldots,t_n\}]\!] = I\!F(\tau[\![t_1]\!])$,
   provided that $\tau[\![t_i]\!] = \tau[\![t_1]\!]$ for
   $i \in \{2,\ldots,n\}$,

5. $\tau[\![(t_1,\ldots,t_n)]\!] = \tau[\![t_1]\!] \times \ldots \times \tau[\![t_n]\!]$,

6. $\tau[\![\langle t_1,\ldots,t_n\rangle]\!] = (\tau[\![t_1]\!])^*$, provided that $\tau[\![t_i]\!] = \tau[\![t_1]\!]$ for $i \in \{2,\ldots,n\}$,

7. $\tau[\![\{\ell_1 \mapsto t_1,\ldots,\ell_n \mapsto t_n\}]\!] = [\ell_1 : \tau[\![t_1]\!],\ldots,\ell_n : \tau[\![t_n]\!]]$ provided $\ell_1,\ldots,\ell_n$ are distinct,

8. consider an application of the form $f(t)$, let f have signature $sign(f)$, then

$$\forall(A,B) \in sign(f) : \tau[\![t]\!] = A \Rightarrow \tau[\![f(t)]\!] = B,$$

9. $\tau[\![\{x : a \mid b\}]\!] = \tau[\![a]\!]$, if there is a type T such that $\tau[\![a]\!] = I\!F(T)$ and if for some value e of type T: $\tau[\![b^x_e]\!] = I\!B$,

10. $\tau[\![(x : a \mid b)]\!] = I\!F(T \times S)$, where T is a type such that $\tau[\![a]\!] = I\!F(T)$ and S a type such that $\tau[\![b^x_e]\!] = S$ for some value e of type T,

11. a value definition of the form $n := t : T$ is well-typed if and only if $\tau[\![t]\!] = \epsilon[\![T]\!]$.

If a type can be derived by these rules a term is well-typed, otherwise it is not well-typed.
□

This definition is in fact a type checking algorithm. Note that the type of a map term is a set of pairs, without the restriction that the first ones are unique.
In rule 6 we assume that f might be polymorphic and therefore we have to assume that there is more than one domain type. Note that singular values are *polymorphic*. For instance:

$$\{(\{\},(\{\}))\}$$

belongs to

$$I\!F(I\!F(T) \times (I\!F(S)^*)$$

for all types T and S.
One could extend the domain of τ to singular values, by extending its range to sets of types , such that τ applied to singular values returns a set of types instead of one type. Such a set of types can be represented by a type expression with type variables. We do not follow this approach here to keep type checking simple. However our type checking is not as powerful as it could be, because we do not type singular values.
Next we will see how to evaluate terms without free variables to values. This is determined by the evaluation function ϵ. Here we assume that

for each function name f, the graph of f can be consulted. This means that we must be able to decide if for two values a and b:

$$(a, b) \in graph(f).$$

Later we will see how this problem can be solved, at least in many

practical cases.

Definition 22.8 The *evaluation function* ϵ maps terms to values according to the following rules. **evaluation function**
If t is a value then $\epsilon[\![t]\!] = t$.
Let $t, t_1, \ldots, t_n$ be a terms from which the value is known.

1. if t is a constant, then $\epsilon[\![t]\!] = t$,
2. if n is a value name defined by $n := t : T$, then $\epsilon[\![n]\!] = \epsilon[\![t]\!]$,
3. $\epsilon[\![(t)]\!] = \epsilon[\![t]\!]$,
4. $\epsilon[\![\{t_1, \ldots, t_n\}]\!] = \{\epsilon[\![t_1]\!], \ldots, \epsilon[\![t_n]\!]\}$, (in fact its normal form),
5. $\epsilon[\![(t_1, \ldots, t_n)]\!] = (\epsilon[\![t_1]\!], \ldots, \epsilon[\![t_n]\!])$,
6. $\epsilon[\![\langle t_1, \ldots, t_n\rangle]\!] = \langle\epsilon[\![t_1]\!], \ldots, \epsilon[\![t_n]\!]\rangle$,
7. $\epsilon[\![\{\ell_1 \mapsto t_1, \ldots, \ell_n \mapsto t_n\}]\!] = \{\ell_1 \mapsto \epsilon[\![t_1]\!], \ldots, \ell_n \mapsto \epsilon[\![t_n]\!]\}$, (in fact its normal form),
8. let t be an application of the form $f(a)$, then $\epsilon[\![f(a)]\!] = b$ if and only if $(\epsilon[\![a]\!], b) \in \mathit{graph}(f)$,
9. let t be a set term of the form $\{x : a \mid b\}$ then (expressed in the meta language):
$$\epsilon[\![t]\!] = \{y \mid y \in \epsilon[\![a]\!] \wedge \epsilon[\![b^x_y]\!] = \mathit{true}\},$$
10. if t is a map term of the form $(x : a \mid b)$, then in the meta language:
$$\epsilon[\![t]\!] = \{(y, z) \mid y \in \epsilon[\![a]\!] \wedge \epsilon[\![b^x_y]\!] = z\}.$$

□

Most of these rules are obvious: it is exactly what we intuitively mean by the language constructs. Note that we require for the evaluations of function applications of the form $f(a)$, that $\epsilon[\![a]\!]$ has to be known before we can determine $\epsilon[\![f(a)]\!]$. This is called *applicative order reduction* in lambda calculus or function programming languages. We will make one exception to this rule for lazy functions: for them we assume that "sufficient" arguments are evaluated. In particular to accommodate *recursion* the selection function *if.then.else.fi* is lazy evaluated. So if we evaluate the term **applicative order reduction**

$$\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z\ \mathit{fi}$$

then we evaluate x first and then we evaluate y only if $\epsilon(x) = \mathit{true}$ and z only if $\epsilon(x) = \mathit{false}$.

We have introduced terms which are used to construct values in an intensional way. One of the main constructions is made by function

application. However we have given only a very limited set of primitive functions. In principle we "have" all functions in UF however we are only able to use them, if we can characterize them with a finite *description*. We use two mechanisms *(lambda) abstraction* and *recursion* to define new functions from primitive or already defined functions.

Abstraction is the well-known way we define functions in mathematics. For instance (in meta language):

$$f(x) = a\ sin(x) + b\ cos^2(x)$$

is a definition of f by giving a term and indicating which name has to be considered as variable (in this case x). In (untyped) lambda calculus this definition is of the form

$$f = \lambda x \bullet (asin(x) + bcos^2(x))$$

An example of abstraction in our languages is:

$$+(x, y) := (x - (0 - y))$$

which is the addition function constructed from the function "$-$" and the constant 0. We may apply this function (in infix notation) as $3 + 5$, which evaluates to 8.

recursion

Recursion is the construction of a function by an *explicit equation*, which means that it is an abstraction with *self-reference*. For instance a recursive equation for the union of sets is:

$$union(x, y) = \mathit{if}\ x = \{\}\ \mathit{then}\ y\ \mathit{else}\ \mathit{ins}\,(pick(x), union(rest(x), y))\ \mathit{fi}$$

The equation is called "explicit" because the value we want to define is given explicitly on the left-hand side of the equation. An example of an implicit equation is:

$$f(x)^2 + f(y)^2 = c.$$

In general it is not sure that such an equation has precisely one solution, it may have many solutions or no solution at all.
Consider for instance the recursive equation

$$f(x) = -f(x + 1)$$

with $x \in \mathbb{Z}$, which has solution

$$f(x) = (-1)^x c \text{ for any } c \in \mathbb{Z}.$$

The next recursive definitions have no solution at all ($x \in \mathbb{N}$):

$$f(x) = f(x) + 1,$$

$$f(x) = f(x + 1) + 1.$$

Now we have defined functions by abstraction of variables from terms, we will give the syntax of function definitions. Each definition ends with the specification of the *signature* of the function. For polymorphic functions we use type variables.

Definition 22.9 The syntax of *function definitions* is: **function definition syntax**

- *function definition* ::=

$$\textit{function name}\underline{(}\textit{(variable)}\underline{)} \underline{:=} \textit{term} \underline{:} \textit{signature},$$

- *function name* $\in FN$,
- *variable* $\in V$,
- *signature* ::= *type expression* $\underline{\Rightarrow}$ *type expression.*

□

In signatures the domain and range types are separated by the $\Rightarrow$ sign. If a type variable occurs, the signature is a set of pairs that belongs to a polymorphic function. If no type variable occurs, the signature is a singleton and belongs to a monomorphic function. Note that a value definition can be considered as a function definition without a domain. Formally the semantics of a signature is defined by the following definition.

Definition 22.10 Let a *signature* $T \Rightarrow S$ be given, where T and S are type expressions. **signature semantics**
If neither T nor S contain type variables, then the semantics of this signature is:

$$\epsilon[\![T \Rightarrow S]\!] = \{(\epsilon[\![T]\!], \epsilon[\![S]\!])\}.$$

If T or T and S contain type variables $\$_1, \ldots, \$_n$, then:

$$\epsilon[\![T \Rightarrow S]\!] = \{(\epsilon[\![A]\!], \epsilon[\![B]\!]) \mid \exists A_1, \ldots, A_n \in TE : A = T^{(\$_1,\ldots,\$_n)}_{(A_1,\ldots,A_n)} \ \wedge \ B = S^{(\$_1,\ldots,\$_n)}_{(A_1,\ldots,A_n)}\}.$$

In case subexpressions of the form $\$_1 \bowtie \$_2$ occur, then $\$_1$ and $\$_2$ may be replaced by compatible tuple types only.
□

Hence all simultaneous substitutions of occurrences of the type variables by "proper" types determine the signature of a function.

We only allow recursion per function construction, so we do not allow for instance that function f is defined using applications of a function g, that is defined using applications of f. This is not a real restriction since we may transform these two *mutual recursive* equations into one recursive equation, as follows. Let

$$f(x) := \textit{term}_1 : A \Rightarrow T$$
$$g(y) := \textit{term}_2 : B \Rightarrow S.$$

We define a function h such that

$$h(x, y) = (f(x), g(y)),$$

by choosing an a and a b such that, for all x and y:

$$f(x) = \pi_1(h(x,b)) \text{ and } g(y) = \pi_2(h(a,y)).$$

Then the definition of h becomes:

$$h(x,y) := (term_1', term_2') : A \times B \Rightarrow T \times S,$$

where in $term_i'$ is term $term_i$ with each occurrence of the form $f(E)$ replaced by $\pi_1(h(E,b))$ and each occurrence of the form $g(E)$ by $\pi_2(h(a,E))$ for expressions E. As an example let:

$$f(x) = l(x, g(b)) \ \wedge \ g(y) = k(y, f(y)).$$

Then we obtain:

$$h(x,y) = (l(x, \pi_2(h(x,b))), k(y, \pi_1(h(y,y)))).$$

To give the semantics of a *function definition* we consider, as for terms, the type rules and evaluation rules.
We start with defining when a function definition is *correctly typed*, i.e. when it has a correct signature. Remember that we consider a function of more than one variable as a function on a product type.

Definition 22.11 A function definition of the form:

$$f(x_1, \ldots, x_n) := t : E_1 \times \ldots \times E_n \Rightarrow E_0$$

where t is a term and $E_0, \ldots, E_n$ are type expressions with possibly type variables $\$_1, \ldots, \$_m$ in the signature, and with no other free variables in t then $x_1, \ldots, x_n$ is *well-typed* if and only if:

well-typed function definition

- all type variables in E_0 also occur in at least one of the type expressions $E_1, \ldots, E_n$,
- for all substitutions of the type variables by type expressions without type variables $A_1, \ldots, A_n$,

  $$B_i = (E_i)^{(\$_1,\ldots,\$_n)}_{(A_1,\ldots,A_n)}$$

  are correct type expressions ($i \in \{0, \ldots, n\}$),
- for all substitutions of the value variables $x_1, \ldots, x_n$ by regular values $e_1, \ldots, e_n$, such that e_i is of type B_i ($i \in \{1, \ldots, n\}$), the value

  $$b = t^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}$$

  has to satisfy:
  - if b does not contain applications of f: $\tau[\![b]\!] = \epsilon[\![B_0]\!]$,
  - if b contains applications of f of the form: $f(c_1, \ldots, c_n)$ then, this expression should be given the type $\epsilon[\![B]\!]$ first.

□

Note that we do not give an algorithm here, but only a definition of correctly typed function constructions. To see how an algorithm could work we give an example. Consider the recursive definition of *union* again. Let A be an arbitrary, non-empty type and let e_1 and e_2 be values of type $\mathbb{F}(A)$. Consider

$$\tau[\![union(e_1, e_2)]\!] = \\ \tau[\![\mathit{if}\ e_1 = \{\}\ \mathit{then}\ e_2\ \mathit{else}\ \mathit{ins}(\mathit{pick}(e_1),\ \mathit{union}(\mathit{rest}(e_1),\ e_2))\mathit{fi}]\!].$$

We derive the type of this term bottom up:

1. $\tau[\![e_2]\!] = \mathbb{F}(A)$ (by assumption),
2. $\tau[\![\mathit{rest}(e_1)]\!] = \mathbb{F}(A)$ (since *rest* is polymorphic),
3. $\tau[\![\mathit{union}(\mathit{rest}(e_1),\ e_2)]\!] = \mathbb{F}(A)$ (by the signature of *union*),
4. $\tau[\![\mathit{pick}(e_1)]\!] = A$ (by polymorphism of *pick*),
5. $\tau[\![\mathit{ins}(\mathit{pick}(e_1),\ \mathit{union}(\mathit{rest}(e_1),\ e_2))]\!] = \mathbb{F}(A)$ (by definition of *ins*),
6. $\tau[\![e_2]\!] = \mathbb{F}(A)$ (given),
7. $\tau[\![e_1 = \{\}]\!] = \mathbb{B}$ (by definition of =),
8. hence the original term has type $\mathbb{F}(A)$, since it fits the signature of *if.then.else.fi*.

Next we define how the graph of a constructed function is determined.

Definition 22.12 A function construction of the form

$$f(x_1, \ldots, x_n) := t : E_1 \times \ldots \times E_n \Rightarrow E_0$$

determines the *graph* of f in the following way: **function graph**

- if the definition is not recursive

$$\mathit{graph}(f) = \{((e_1, \ldots, e_n), \epsilon[\![t^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!]) \mid (e_1, \ldots, e_n) \in U\},$$

- if the definition is recursive , then f is a solution of the equation with unknown graph g:

$$g = \{((e_1, \ldots, e_n), \epsilon_g[\![t^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!]) \mid (e_1, \ldots, e_n) \in U\},$$

  where ϵ_g is the evaluation function ϵ modified in such a way that every evaluation of the form $\epsilon[\![f(e_1, \ldots, e_n)]\!]$ is replaced by c if and only if $(e_1, \ldots, e_n), c) \in \mathit{graph}(g)$.

□

Note that we did not specify which function should be chosen, in case there are more solutions to the equation in g. In chapter 24 we will see which one we recommend to choose.

Another way to define the graph of f is by means of a λ-expression. Define a function F as follows:

$$F = \lambda g \bullet \lambda x_1, \ldots, x_n \bullet t_g^f$$

and let function f be a fixed point of F, i.e. $F(f) = f$. (Note that we mix here the specification language with the meta language.)

As an application of these primitive functions we consider the logical functions $\vee$, $\wedge$, $\Rightarrow$ and $\neg$ which can be constructed from these primitives. In part I we gave the standard definitions for these functions. Here we give them for the *three-valued logic* with *truth values* in $\mathbb{B}_\perp$ (i.e. *true*, *false* and $\perp$). We will use them in the next chapter.
We start with $\Rightarrow$, it satisfies the equation:

$$\begin{aligned} x \Rightarrow y := \ & \textit{if } x = \perp \textit{ then} \\ & \quad \textit{if } y \textit{ then true else } \perp \textit{fi} \\ & \textit{else} \\ & \quad \textit{if } x \textit{ then } y \textit{ else true fi} \\ & \textit{fi} : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}. \end{aligned}$$

(Note the different meanings of $\Rightarrow$.)
This gives the following truth table for $\Rightarrow$:

$\Rightarrow$			y	
		true	*false*	$\perp$
x	*true*	*true*	*false*	$\perp$
	false	*true*	*true*	*true*
	$\perp$	*true*	$\perp$	$\perp$

With this function we can define the other logical functions in a well-known way:

$$\begin{aligned} \neg x &:= x \Rightarrow \textit{false} : \mathbb{B} \Rightarrow \mathbb{B}, \\ x \vee y &:= \neg x \Rightarrow y : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}, \\ x \wedge y &:= \neg(\neg x \vee \neg y) : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}. \end{aligned}$$

The truth tables for these functions can be derived easily, for example

$\vee$			y	
		true	*false*	$\perp$
x	*true*	*true*	*true*	*true*
	false	*true*	*false*	$\perp$
	$\perp$	*true*	$\perp$	$\perp$

In this way we define in the language all other functions we need. Often we want functions to be strict. If we have defined a function, for example

$f : \$_1 \Rightarrow \$_2$, that is non-strict and we want to make it strict we simply define:

$$\tilde{f}(x) := \textit{if } x = \bot \textit{ then } \bot \textit{ else } f(x) \textit{ fi} : \$_1 \Rightarrow \$_2.$$

(In examples we often "forget" this modification, because it makes the specification less readable.)

Chapter 23

Declarative part of the language

In the foregoing chapters we introduced the *constructive* part of the specification language: finite mathematical values and function definitions. Specifying systems in this way is on one hand nice: we get constructs that can be evaluated by a machine, so we can *simulate* the systems we define. The only problem is recursive constructions, where we have to find a solution (cf. the next chapter). On the other hand the *expressive comfort* is limited. In this chapter we introduce *declarative expressions* that cannot be executed. They are meant for the systems engineer who has to transform them into constructive expressions. Consider for instance the construction of the *union* function we have seen before. A more natural and better understandable specification is:

- name : *union*
- signature : $\mathbb{F}(T) \times \mathbb{F}(T) \Rightarrow \mathbb{F}(T)$
- predicate : $\forall x, y : \mathbb{F}(T) \bullet \forall z : T \bullet$

$$z \in union(x, y) \Leftrightarrow z \in x \vee z \in y$$

This is an example of a *function declaration*. This concept will be defined later. We see here a *predicate* that involves quantification over a type. Up to now we only considered quantification over sets: the set term and the map term. The Boolean functions *forall* and *exists* are examples (cf. Toolkit). Note that in general, such a quantification over a type is not computable, i.e. there is no algorithm for it. However for expressive comfort we will introduce them.

Another reason for introduction of predicates and quantification over types is that we wish to introduce *restricted tuple types*, which are called *schemas*. We have seen examples of schemas in part I. Schemas are used for the specification of complex classes and processor relations.

23.1 Predicates and function declarations

A *predicate* generalizes the concept of a Boolean type term.

predicate syntax

Definition 23.1 Given a syntax base and the syntax defined so far, *predicates* are defined by:

- $$\begin{array}{lll} \textit{predicate} & ::= & \textit{bool} \mid \neg\, \textit{predicate} \mid (\textit{predicate}\ \theta\ \textit{predicate}) \mid \\ & & \textit{quantor}\ (\textit{variable}) : \textit{domain} \bullet \textit{predicate} \\ \textit{domain} & ::= & \textit{type expression} \end{array}$$
- $\theta \in \{\vee, \wedge, \Rightarrow\}$
- $\textit{quantor} \in \{\forall, \exists\}$
- $\textit{bool} ::= \textit{Boolean term}$
- $\textit{variable} \in V$

A Boolean term is a term with type $\mathbb{B}$.
The functions $\neg$, $\vee$, $\wedge$ and $\Rightarrow$ are defined in chapter 22.
□

The quantors ($\forall$ and $\exists$) have *scope rules* in a similar way as we defined them in definition 22.6. Predicates without free variables have semantics: the function ϵ assigns one of the values *true*, *false* or $\perp$ to them. Therefore they can be used as Boolean values.

To define which predicates are *well-typed* we proceed along the same lines as we did for function constructions.

well-typed predicates

Definition 23.2 Predicates without free (value) variables are *well-typed* if the following rules hold.
Consider first the case without type variables:

- if a predicate p is a Boolean term then $\epsilon[\![p]\!] = \mathbb{B}$,
- if p is a well-typed predicate then

  $$\forall\, x_1, \ldots, x_n : T \bullet p$$

  and

  $$\exists\, x_1, \ldots, x_n : T \bullet p$$

  are well-typed if and only if x is the only free variable of predicate p and for any value e of type T: $p^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}$ is well-typed.

In case any domain contains type variables, these rules should apply for all substitutions of these type variables with type expressions without type variables.
□

The semantics of predicates in case of no free variables and no type variables is given below. We extend the evaluation function ϵ defined in definition 22.8 to predicates.

Definition 23.3 Predicates without free value variables and without type variables are *evaluated* according to the following rules. Assume predicate p is well-typed.

1. if predicate p is a Boolean term then $\epsilon(p)$ is already defined,
2. $\epsilon[\![\neg p]\!] = \epsilon[\![\neg(\epsilon[\![p]\!])]\!]$,
3. for a predicate of the form: $p\,\theta\,q$ where $\theta \in \{\vee, \wedge, \Rightarrow\}$:

$$\epsilon[\![p\,\theta\,q]\!] = \epsilon[\![\epsilon[\![p]\!]\;\theta\;\epsilon[\![q]\!]]\!],$$

4. if p is a predicate of the form

$$\forall x_1, \ldots, x_n : T \bullet q$$

then

- $\epsilon(p)$ is *true* if for all values $e_1, \ldots, e_n$ of type T: $\epsilon[\![q^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!] = \mathit{true}$,
- $\epsilon(p)$ is *false* if there is at least one row $(e_1, \ldots, e_n)$ of type T^* such that $\epsilon[\![q^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!] = \mathit{false}$,
- in all other cases $\epsilon(p) = \bot$,

5. if p is a predicate of the form

$$\exists x_1, \ldots, x_n : T \bullet q$$

then

- $\epsilon(p)$ is *true* if there is at least one row $(e_1, \ldots, e_n)$ of type T^* such that $\epsilon[\![q^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!] = \mathit{true}$,
- $\epsilon(p)$ is *false* if for all values $e_1, \ldots, e_n$ of type T: $\epsilon[\![q^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}]\!] = \mathit{false}$,
- in all other cases $\epsilon(p) = \bot$.

□

We apply predicates to *function declarations*. We have seen how useful it is to have another mechanism to describe a function in addition to the function definition.

Definition 23.4 The syntax of a *function declaration*, given a syntax base and the syntax introduced so far, is: **function declaration syntax**

function declaration ::= *function name* :: *signature* :: *predicate*

□

We have to define what we mean by a *well-typed* function declaration and what the *meaning* of a well-typed function declaration is. This is done in the next definitions.

well-typed function declaration

Definition 23.5 A function declaration is *well-typed* if and only if

- the predicate is well-typed,
- the predicate does not contain free (value) variables,
- the type variables in the predicate also occur in the signature,
- the signature is well-typed (i.e. all type variables occur at least in the domain type).

□

Definition 23.6 The meaning of a well-typed function declaration of the form

$$f :: D \Rightarrow R :: p$$

where D and R are type expressions, possibly with type variables $\$_1, \ldots, \$_n$, is the set of all (mono- or polymorphic) functions g such that:

- $sign(g) =$

$$\{(\epsilon[\![D^{(\$_1,\ldots,\$_n)}_{(T_1,\ldots,T_n)}]\!], \epsilon[\![R^{(\$_1,\ldots,\$_n)}_{(T_1,\ldots,T_n)}]\!]) \mid T_1, \ldots, T_n \in TE\},$$

- for all suitable type expressions $T_1, \ldots, T_n \in TE$:

$$\epsilon[\![p^{(f,\$_1,\ldots,\$_n)}_{(g,T_1,\ldots,T_n)}]\!] = true.$$

(Note that g is used as a function name and that it is assumed that $graph(g)$ is known and that TE is the set of type expressions without type variables.)
□

So the meaning of a function declaration is the set of all functions that fit into its signature and have the property expressed by the predicate. Note that this set may be empty as in the next example:

$$f :: \mathbb{N} \Rightarrow \mathbb{B} :: \forall x : \mathbb{N} \bullet (3 \times (x \; div \; 3) = x \Rightarrow f(x) = true)$$
$$\wedge$$
$$(5 \times (x \; div \; 5) = x \Rightarrow f(x) = false).$$

It is impossible to give a function definition for such a function because we have to indicate explicitly for each argument a unique value and for all multiples of 15 we have here two different values. It may also happen that a function declaration denotes a set with many elements, for instance

$$f :: \mathbb{N} :: \forall x : \mathbb{N} \bullet f(x) \leq f(x+1).$$

It is the task of the systems engineer to prove that there is at least one function in the set. If there is more than one function the systems engineer leaves the choice to a constructor of the system.

23.2 Schemas and scripts

Schemas play an important role in models because they are used to define complex classes on the one hand and processor relations for processors on the other hand. The complexes can be represented as *tuples* belonging to a schema and the firing rules of processor relations are represented by tuples of the schema that belongs to the processor. The schemas for processor relations may contain type variables to express that we have a *generic* processor definition that can be used in different locations in an actor model with different type substitutions. Because schemas are so important there is a set of *operators* on schemas to be able to construct a schema stepwise, out of more simple schemas. The syntax of schemas is given in the next definition.

Definition 23.7 Given a syntax base a *schema* has the following syntax

schema expression syntax

- *schema* ::= [*schema signature* | *predicate*]
- *schema signature* ::= ⟨ *variable* : *type expression*⟩
- *variable* $\in V \cap L$

All free variables of the predicate have to occur in the schema signature as well.
□

Note that we use here that *attributes* may also be *variables*. They have a different meaning depending on their role in expressions. The semantics of a schema is defined below.

Definition 23.8 Consider a schema of the form:

$$[x_1 : T_1, \ldots, x_n : T_n \mid p].$$

Then:

$$\epsilon[\![[x_1 : T_1, \ldots, x_n : T_n \mid p]]\!] = \{z : \epsilon[\![[x_1 : T_1, \ldots, x_n : T_n]]\!] \mid \epsilon[\![p^{(x_1,\ldots,x_n)}_{(\pi_1(z),\ldots,\pi_n(z))}]\!]\}.$$

The value z belongs only to the set, if the predicate evaluates to *true*. (So if it is $\perp$ then z does not belong to the set.) The set of all schemas without type variables is called the *schema universe* and is denoted by SU.

schema universe

□

So a schema is in fact a *restricted set of tuples*.

Definition 23.9 A schema without type variables, of the form:

$$[x_1 : T_1, \ldots, x_n : T_n \mid p]$$

well-typed schema

is *well-typed* if for all values $e_1, \ldots, e_n$ of types $T_1, \ldots, T_n$ respectively,

$$p^{(x_1,\ldots,x_n)}_{(e_1,\ldots,e_n)}$$

are well-typed.
If there are type variables involved, the assertion should hold for all, possible substitutions of the type variables by type expressions without type variables.
□

Next we introduce schema expressions and schema definitions. Afterwards we give some examples that illustrate the use of these expressions. This part of the language incorporates most of the *schema calculus* of Z.

schema expression syntax
schema definition syntax

Definition 23.10 A *schema expression* and a *schema definition* have the following syntax:

- *schema expression* ::= *schema* | (*schema expression*) |
  schema name(⟨ *type expression* ⟩) |
  ¬ *schema expression* |
  schema expression θ *schema expression* |
  schema expression \ (⟨ *variable* ⟩) |
  schema expression ↾ (⟨ *variable* ⟩) |
  quantor variable | *predicate* • *schema expression*
- *schema definition* ::=
  schema name[(⟨*type variable*⟩)] ≔ *schema expression*
- $\theta \in \{\vee, \wedge, \Rightarrow\}$
- *quantor* $\in \{\forall, \exists\}$
- *variable* $\in V \cap L$

The following conditions should hold:

- the variables left of the symbols \ and ↾ should occur in the schema signatures of the schema expression,
- the variable behind a quantor should appear in a signature of the schema expression with the same type expression, and the predicate should have no free variables or type variables,
- the schema expression in a schema definition may contain type variables, however these type variables must appear also on the left hand side of the ":=" symbol,
- schema definitions may not be recursive.

□

A schema expression, in case there are no type variables, determines a schema. For all the expressions we will define the semantics below. A schema definition with type variables defines a *schema-valued function* over a Cartesian product of TE.
For example:

$$h(\$) := [x : \$, y : I\!F(\$) \mid x \in y]$$

defines a *function* h. For every $T \in TE$ the function value $h(T)$ is obtained by substituting T for \$. If there are no type variables on the right hand side of a schema definition, then the schema name is not allowed to have type variables, in which case it is used as a shorthand for a schema expression, like a type definition. The requirement that schema definitions are not recursive means that the schema name on the left hand side is not allowed to occur on the right hand side.

In order to define the semantics of schema expressions and schema we introduce a *syntactical transformation function* that maps schema expressions to schemas. This function is called δ. So the semantics of a schema expression is in fact the *composition* of ϵ and δ, first we transform a schema expression into a schema (δ) and then we apply the evaluation function (ϵ).

syntactical transformation function δ

Definition 23.11 The semantics of a schema expression without type variables is given by the function δ. Let s be a schema expression.

1. if s is a schema then $\delta[\![s]\!] = s$,
2. $\delta[\![(s)]\!] = s$,
3. $\delta[\![\neg[x_1 : T_1, \ldots, x_n : T_n \mid p]]\!] = \delta[\![[x_1 : T_1, \ldots, x_n : T_n \mid \neg p]]\!]$,
4. $\delta[\![[x_1 : T_1, \ldots, x_n : T_n \mid p] \; \theta \; [y_1 : S_1, \ldots, y_m : S_m \mid q]]\!] =$

   $$[z_1 : R_1, \ldots, z_\ell : R_\ell \mid p \theta q],$$

   where

   - $\theta \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$,
   - $\forall i, j : x_i = y_j \Rightarrow T_i = S_j$,
   - $\{z_1, \ldots, z_\ell\} = \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$,
   - $\forall i, j, k : z_k = x_i \Rightarrow R_k = T_i \wedge z_k = y_j \Rightarrow R_k = S_j$,
5. if $n \geq m$: $\delta[\![[x_1 : T_1, \ldots, x_n : T_n \mid p] \setminus (x_1, \ldots, x_m)]\!] =$

   $$[x_{m+1} : T_{m+1}, \ldots, x_n : T_n \mid \exists x_1 : T_1 \bullet \ldots \exists x_m : T_m \bullet p],$$

6. if $n \geq m$: $\delta[\![[x_1 : T_1, \ldots, x_n : T_n \mid p] \restriction (x_1, \ldots, x_m)]\!] =$

   $$[x_1 : T_1, \ldots, x_m : T_m \mid \exists x_{m+1} : T_{m+1} \bullet \ldots \exists x_n : T_n \bullet p\},$$

7. $\delta[\![\forall x_1 : T \mid q \bullet t[x_1 : T_1, \ldots, x_n : T_n \mid p]]\!] =$

$$[x_2 : T_2, \ldots, x_n : T_n \mid \forall x_1 : T \bullet q \Rightarrow p],$$

8. $\delta[\![\exists x_1 : T \mid q \bullet t[x_1 : T_1, \ldots, x_n : T_n \mid p]]\!] =$

$$[x_2 : T_2, \ldots, x_n : T_n \mid \exists x_1 : T \bullet q \ \wedge \ p].$$

Variables may be rearranged first.
The semantics of schemas is given in definition 23.8.
□

schema definition semantics

Definition 23.12 The *semantics* of a *schema definition* of the form

$$f(\$_1, \ldots, \$_n) := s$$

(where $n \geq 0$) is a function, also denoted by f, such that

$$f \in TE^n \ \rightarrow SU,$$

where TE is the set of all type expressions without free variables and SU the schema universe and where for $T_1, \ldots, T_n \in TE$:

$$f(T_1, \ldots, T_n) = \delta[\![s^{(\$_1,\ldots,\$_n)}_{(T_1,\ldots,T_n)}]\!]$$

if the right-hand side is defined.
□

We introduce the concept of *equality* for schema expressions.

schema equality

Definition 23.13 Two schema expressions s_1 and s_2 are said to be *equal*, (notation $s_1 = s_2$) if and only if:

- $\epsilon[\![\delta[\![s_1]\!]]\!]$ and $\epsilon[\![\delta[\![s_2]\!]]\!]$ are defined,
- $\epsilon[\![\delta[\![s_1]\!]]\!] = \epsilon[\![\delta[\![s_2]\!]]\!]$.

□

We will elucidate these definitions with some examples.
Consider the schema definitions

$$\begin{aligned} s_1 &:= [x : I\!N, y : I\!N \mid x \leq y] \\ s_2 &:= [y : I\!N, z : I\!N \mid y \leq z] \\ s_3 &:= [x : I\!N, y : I\!N, z : I\!N \mid x + z = y] \end{aligned}$$

Obviously:

- $\{x \mapsto 1, y \mapsto 3\}$ belongs to s_1 and $\{x \mapsto 1, y \mapsto 2, z \mapsto 1\}$ to s_3,
- $s_1 \wedge s_2 = [x : I\!N, y : I\!N, z : I\!N \mid x \leq y \wedge y \leq z]$,
- $s_1 \Rightarrow s_2 = [x : I\!N, y : I\!N, z : I\!N \mid x \leq y \Rightarrow y \leq z]$,

- $\{x \mapsto 1, y \mapsto 0, z \mapsto 0\}$ belongs to $s_1 \Rightarrow s_2$,
- $\{x \mapsto 1, y \mapsto 3, z \mapsto 4\}$ belongs to $s_1 \wedge s_2$,
- $s_3 \backslash (z) = [x : I\!N, y : I\!N \mid \exists z : I\!N \bullet x + z = y]$,
- hence (by the properties of natural numbers): $s_3 \backslash (z) = s_1$,
- $s_3 \upharpoonright (z) = [z : I\!N \mid \exists x : I\!N \bullet \exists y : I\!N \bullet x + z = y]$,
- $\forall z : I\!N \mid z \; mod \; 2 = 0 \bullet s_2 =$
  $[y : I\!N \mid \forall z : I\!N \bullet z \; mod \; 2 = 0 \Rightarrow y \leq z] =$
  $[y : I\!N \mid y = 0]$ (by properties of $I\!N$),
- $\exists z : I\!N \mid true \bullet s_3 = [x : I\!N, y : I\!N \mid \exists z : I\!N \bullet x + z = y] = s_1$.

Note that a schema with predicate *true* is in fact a tuple type, i.e.

$$[x_1 : T_1, \ldots, x_n : T_n \mid true] = [x_1 : T_1, \ldots, x_n : T_n].$$

As a consequence we have:

$$[x_1 : T_1, \ldots, x_n : T_n \mid true] \;\wedge\; [y_1 : S_1, \ldots, y_n : S_n \mid true] = [x_1 : T_1, \ldots, x_n : T_n] \bowtie [y_1 : S_1, \ldots, y_n : S_n].$$

Another property is:

$$\exists x : T \mid p \bullet [x : T, y_1 : S_1, \ldots, y_n : S_n \mid q] = ([x : T \mid p] \wedge [x : T, y_1 : S_1, \ldots, y_n : S_n \mid q]) \backslash (x)$$

(The proof is an exercise.)
These properties show that our language is redundant, but that is a consequence of the wish to have great expressive comfort.

Next we introduce *scripts*. A script is a coherent set of type definitions, function definitions, function declarations and schema definitions. **script**

We required several times that we do not allow recursion in type definitions, schema definitions, and function specifications. For function constructions we allow a limited form of recursion.

To formalize what we mean, we will require that all definitions and declarations in a script, can be given a *rank*, (i.e. a natural number) such that all names used in this definition or declaration have a definition or declaration with a lower rank. For function definitions we make an exception: they may use names with equal rank also (i.e. the name of the definition may be used in the definition). This means that we define according to the principle: *define before use*, with an exception for function construction. In the next definition we give the syntax of a script.

Definition 23.14 A *script* has the following *syntax*: **script syntax**

- *script* ::= *line* | *line* ; *script*

- *line* ::= *type definition* | *value definition* |
  function definition | *function declaration* |
  schema definition

such that there is a function, that assigns a natural number n to every line ℓ , with the property that every name occurring in the definition part of the line is defined itself in a line with a number smaller or equal to n in case the line is a function definition and smaller than otherwise. □

In the table format of the language we allow all kind of definitions in a *schema body*, i.e. the part where the predicate is written. A definition in a schema body is considered to be *local* to that schema.

Chapter 24

Methods for function construction

We address four problems: *correctness* of recursive constructions, *finding* them, *transforming* them into easier to handle constructions and the *transformation* of function *declarations* into *constructions*. The only difficult part of function construction is recursion, therefore we focus on that aspect here. In some cases the systems engineer will declare a function first, then he transforms this declaration into a construction and finally he transforms this construction into an *algorithm*. This is the longest path. It is also possible that he starts with a construction and that it turns out that the construction can be executed fast enough.

24.1 Correctness of recursive constructions

As said before, a recursive construction is the definition of a function by means of an *explicit equation*. In order to have a *correct* recursive construction we have to answer three questions:

- *existence*: is there a solution?
- *unicity*: is there only one solution and if there are more which one should we choose?
- *computability*: is the solution computable, i.e. is there an algorithm to compute for an arbitrary argument the corresponding function value?

Note that the first question is undecidable, so there is no algorithm to solve this problem.

In order to answer these questions we introduce first another view on recursive equations. Consider for example the recursive equation:

$$union(x, y) :=$$
$$\quad \textit{if } x = \{\} \textit{ then } y \textit{ else ins } (pick(x), union(rest(x), y))\ \textit{fi} : \$ \times \$ \Rightarrow \$.$$

Another view on this equation is that the term that defines the function *union*, specifies a function in $U \nrightarrow U$ with *union* as argument.

In lambda calculus this function can be expressed as:

$$\lambda f \bullet \lambda x, y \bullet \mathit{if}\ x = \{\}\ \mathit{then}\ y\ \mathit{else}\ \mathit{ins}\ (\mathit{pick}(x), f(\mathit{rest}(x), y))\ \mathit{fi}.$$

(Note that we do not allow such functions in the specification language, but it is allowed in the meta language.) We call this function the *recursion operator* of the equation. Then we may rephrase the problem as: find a solution of the equation in f with recursion operator F:

recursion operator

$$f = F(f).$$

So *union* is a *fix point* of F and therefore:

$$f = F(f) = F^2(f) = \ldots = F^n(f).$$

It is in many cases possible to compute, for some argument x, $f(x)$ by *iterated application* of F, i.e. by $F^n(f)(x)$. It turns out that we can compute $F^n(f)(x)$ *without* knowing f. As an example consider the multiplication function *mult*:

$$\begin{aligned} \mathit{mult}\ (x, y) := \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ y + \mathit{mult}\ (x - 1, y)\ \mathit{fi} \\ : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \end{aligned}$$

For a specific argument ($\mathit{mult}(2,5)$) iterated application works as follows:

$$\begin{aligned} &\mathit{mult}\ (2,5) = \\ &\mathit{if}\ 2 = 0\ \mathit{then}\ 0\ \mathit{else}\ 5 + \mathit{mult}(1,5)\mathit{fi} = \\ &5 + \mathit{mult}\ (1,5) = \\ &5 + (\mathit{if}\ 1 = 0\ \mathit{then}\ 0\ \mathit{else}\ 5\ + \mathit{mult}(0,5)\mathit{fi}) = \\ &5 + 5 + \mathit{mult}(0,5) = \\ &5 + 5 + (\mathit{if}\ 0 = 0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{mult}(-1,5)\mathit{fi}) = \\ &10 \end{aligned}$$

Note that $\mathit{mult}(-1,5)$ does not have to be evaluated: it would have given $\perp$, since -1 is not in $\mathbb{N}$.

Next we consider the *method of successive approximations*, which is a general method to solve fix point equations. In the rest of this section we do not consider signatures of function definitions and we will identify graphs of functions with their names.
We start with the definition of some technical concepts.

Definition 24.1

- A *partial ordering* on $U \to U$ is defined by:

$$\forall f, g \in U \to U : f \subset g \Leftrightarrow \forall x \in U :\ f(x) = g(x) \vee f(x) = \perp.$$

- Let $\langle f_0, f_1, \ldots \rangle$ be a sequence of functions in $U \to U$. The sequence is called *monotonous* if and only if: **monotonous sequence**

$$\forall n \in \mathbb{N} : f_n \subset f_{n+1}.$$

- The *limit* of a monotonous sequence, denoted by $lim_{n \in \mathbb{N}} f_n$, is: **limit of a monotonous sequence**
  $(lim_{n \in \mathbb{N}} f_n)(x) = \bot$, if $\forall n \in \mathbb{N} : f_n(x) = \bot$,
  $f_k(x)$, for some k such that $f_k(x) \neq \bot$.
  (Hence for all k: $f_k \subset lim_{n \in \mathbb{N}} f_n$.)

- A function $F \in U \nrightarrow U$ is called *monotonous* if and only if: **monotonous function**

$$\forall f, g \in \mathrm{dom}(F) : f \subset g \Rightarrow F(f) \subset F(g).$$

- A function $F \in U \mapsto U$ is called *stable* if and only if for all monotonous sequences $\langle f_0, f_1, \ldots \rangle$: **stable function**

$$F(lim_{n \in \mathbb{N}} f_n) \subset lim_{n \in \mathbb{N}} F(f_n).$$

□

The method of successive approximations constructs a monotonous sequence of functions $\langle f_0, f_1, \ldots \rangle$ such that the limit is the fix point of the recursive equation.

Theorem 24.1 Let $F \in U \mapsto U$ be *monotonous* and *stable* and let sequence $\langle f_0, f_1, \ldots \rangle$ satisfy:

- $\forall x \in U : f_0(x) = \bot$,
- $\forall n \in \mathbb{N} : f_{n+1} = F(f_n)$.

Then $f^* = lim_{n \in \mathbb{N}} f_n$ has the property:

$$f^* = F(f^*).$$

The components of the sequence are called *successive approximations*. **successive approximations**

Proof. From the monotonicity we derive by induction, that $\forall n \in \mathbb{N} : f_n \subset f_{n+1}$. Note that $f_0 \subset f_1$. Assume $f_n \subset f_{n+1}$.
Then $f_{n+1} = F(f_n) \subset F(f_{n+1}) = f_{n+2}$.
So the sequence is monotonous and for all $k \in \mathbb{N}$ we have $f_k \subset f^*$.
Hence

$$f_k \subset f_{k+1} = F(f_k) \subset F(f^*)$$

and so

$$f^* \subset F(f^*).$$

From the stability we derive: $F(f^*) \subset lim_{n \in \mathbb{N}} F(f_n)$.
Since $lim_{n \in \mathbb{N}} f_n = lim_{n \in \mathbb{N} \setminus \{0\}} f_n$ and $F(f_n) = f_{n+1}$ we have

$$F(f^*) \subset f^*.$$

Combining these cases gives the desired result.
□

method of successive approximations

The approach of solving a recursive equation by means of such a sequence is called the *method of successive approximations*. Note that we do not have to compute these functions completely (which is impossible) but that we compute them for sufficiently many arguments in case we want to compute the value of the fix point function for one given argument.

We call the method of successive approximations *applicable* for a recursive equation if the recursion operator F is monotonous and stable.

Theorem 24.2 Under the conditions of the former theorem, the solution f^* is the *smallest* solution of the equation in the sense that, if g is another fix point of F then $f^* \subset g$.

Proof. We prove this by induction. Of course $f_0 \subset g$. Assume that $f_n \subset g$. Since F is monotonous we have $F(f_n) \subset F(g)$. Because g is a fix point and by the definition of f_{n+1}, we have $f_{n+1} \subset g$. Hence $\forall n \in I\!N : f_n \subset g$ and therefore $f^* \subset g$.
□

There is a good reason to take always the least fix point f^* because all other solutions g have the property

$$\forall x : f^*(x) \neq \perp \Rightarrow f^*(x) = g(x),$$

so f^* is the only "sure" solution. The next theorem shows that f^* is "reached" by successive approximations in finitely many steps, for each argument.

Theorem 24.3 Under the conditions of the theorem 24.1 we have:

$$\forall x \in U : \exists n \in I\!N : f^*(x) = F^n(f_0)(x).$$

Proof. The proof is an immediate consequence of: $f_n = F^n(f_0)$ and the definition of the limit.
□

iterated application

linear recursive functions

So if the method of successive approximations is applicable, we can compute the function value of the least fix point by *iterated application* of the recursion operator. We will apply this result to an important special case, *linear recursive functions*. Here we give a *sufficient* condition for applicability.

Theorem 24.4 Consider the equation for a *linear recursive function*:

$$f(x) := \mathit{if}\ b(x)\ \mathit{then}\ a(x)\ \mathit{else}\ h(x, f(g(x)))\ \mathit{fi}$$

where a, b, h, g are strict functions.
Then the method of successive approximations is *applicable*.

Proof. Let F be the recursion operator of the equation. First we show the *monotonicity* of F. Let p and q be functions such that $p \subset q$ and

let $x \neq \perp$ be given. If $F(p)(x) = \perp$ then nothing has to be proven, otherwise either $b(x) = \mathit{true}$ or $p(g(x)) \neq \perp$ (use strictness of h). In the latter case $p(g(x)) = q(g(x))$, since $p \subset q$. So in both cases we have $F(p)(x) = F(g)(x)$, which proves the monotonicity of F.

The next step is to verify the *stability*. Let $\langle s_0, s_1, \ldots \rangle$ be a monotonous sequence and $s^* = lim_{n \in \mathbb{N}} s_n$. If $F(s^*)(x) = \perp$ nothing has to be proven, otherwise either $b(x) = \mathit{true}$ or $s^*(g(x)) \neq \perp$. In the first case $F(s^*)(x) = F(s_n)(x) = a(x)$ (for all $n \in \mathbb{N}$).
In the latter case there is an $n \in \mathbb{N}$ such that $s_n(g(x)) \neq \perp$, since the sequence is monotonous. Hence since $s_n \subset s^*$, we have
$s^*(g(x)) = s_n(g(x))$ and so $F(s^*)(x) = F(s_n)(x)$.
So we have shown that

$$\forall x \in U : F(s^*)(x) \neq \perp \Rightarrow \exists n \in \mathbb{N} : F(s^*)(x) = F(s_n)(x)$$

hence $F(s^*) \subset lim_{n \in \mathbb{N}} F(s_n)$.
□

For this case we can derive a more detailed result.

Theorem 24.5 Let F be defined as in the former theorem. If for some $x \in U$ the following properties hold:

$$b(x) = b(g(x)) = \ldots = b(g^{n-1}(x)) = \mathit{false} \ \wedge \ b(g^n(x)) = \mathit{true},$$

then:

$$f^*(x) = h(x, h(g(x), \ldots, h(g^{n-1}(x), a(g^n(x))) \ldots))$$

and for $k > n$: $F^k(f_0)(x) = f^*(x)$.

Proof. Since $b(x) = \mathit{false}$ we have:

$$f^*(x) = F(f^*)(x) = h(x, f^*(g(x))).$$

By iterated application we obtain:

$$f^*(x) = F(f^*)(x) = h(x, h(g(x), \ldots, h(g^{n-1}(x), f^*(g^n(x))) \ldots)) \quad (*)$$

Since $b(g^n(x)) = \mathit{true}$ we have $f^*(g^n(x)) = a(g^n(x))$. If we substitute this in $(*)$ we obtain the first assertion.
Similarly

$$f_{n+1}(x) = h(x, h(g(x), \ldots, h(g^{n-1}(x), f_1(g^n(x))) \ldots)).$$

Since $b(g^n(x)) = \mathit{true}$ we have $f_1(g^n(x)) = a(g^n(x))$.
So $f_{n+1}(x) = F^{n+1}(f_0) = f^*(x)$.
□

Note that we used the laziness of *if.then.else.fi* here. Next we consider a class of recursive equations for which there is always a solution that can be obtained by one of the methods discussed above. The functions defined in this way are called *primitive recursive functions*. They are very important because most of the functions we encounter in practice are primitive recursive (cf. Toolkit).

primitive recursive function construction

Definition 24.2 A function construction is *primitive recursive* if it is of the form:

$$f(x,y) := \textit{if } x = 0 \textit{ then } a(x,y) \textit{ else } h(x,y,f(x-1,y)) \textit{ fi}$$

where a and h are given strict functions and the type of x is $\mathbb{N}$.
□

Primitive recursive functions satisfy the conditions of theorem 24.4. To verify this let $b(z) := (\pi_1(z) = 0)$, $z = (x,y)$ and $g(z) := (\pi_1(z) - 1, \pi_2(z))$. Then we may transform the equation for f as:

$$f(z) := \textit{if } b(z) \textit{ then } a(z) \textit{ else } h(z, f(g(z))) \textit{ fi}.$$

Theorem 24.6 Consider a primitive recursive equation in f as defined above. This recursive equation has a *unique solution*.

Proof. Let f and g be two solutions. Fix some y. Clearly: $f(0,y) = a(0,y) = g(0,y)$. Assume for some $n \in \mathbb{N}$ that $f(n,y) = g(n,y)$. Then:

$$f(n+1,y) = h(n+1,y,f(n,y)) = h(n+1,y,g(n,y)) = g(n+1,y).$$

Hence we have shown by induction, that there is at most one solution. By iterated application we find:

$$f(n,y) = h(n,y,h(n-1,y,\ldots,h(1,y,a(0,y)))).$$

□

There are many applications of this theorem. For instance the multiplication function *mult* has a unique solution according to this theorem. There are different syntactical forms of these theorems, for example the functions a, b, h and g could have been given by their defining terms instead of an application.

24.2 Derivation of recursive constructions

One of the major problems of constructive specifications is to find a correct and easy to understand recursive construction for a function f. In many cases we use the following approach:

- determine the signature $S \Rightarrow T$ of the function f,
- determine a subset B of S , so B is a value of type $\mathbb{F}(S)$, on which the function is known (note that B is often given in the form of a Boolean function on b with signature $S \Rightarrow \mathbb{B}$),
- determine a function a with the same signature as f that coincides with f on B: $\forall x \in B : f(x) = a(x)$,

- determine a set-valued function R with signature $S \Rightarrow \mathbb{F}(S)$ with the meaning that the value $f(x)$ can be expressed by means of the values $f(y)$ for $y \in R(x)$, in case $x \notin B$,
- determine a function h that tells how to compute $f(x)$ from the values given by $R(x)$: $f(x) = h(x, \{(y, f(y)) \mid y \in R(x)\})$,
- create an equation of the form:

  $$f(x) := \mathit{if}\ x \in B\ \mathit{then}\ a(x)\ \mathit{else}\ h(x, \{(y, f(y)) \mid y \in R(x)\})\mathit{fi}.$$

  Note that this is not an expression in the language, but a *template* for such expressions.

dynamic programming

This method is very similar to the technique called *dynamic programming* and the technique to construct *differential equations* to describe physical phenomena.

We will illustrate these steps with some examples. The first example is the well-known the Fibonacci sequence. The function fib computes the n-th value of this sequence. We follow the steps:

- the signature of fib is $\mathbb{N} \Rightarrow \mathbb{N}$, so $S = \mathbb{N}$,
- $B = \{0, 1\}$,
- $\forall x \in B:\ fib(x) = 1$,
- $\forall x \in S : R(x) = \{x - 1, x - 2\}$,
- $h(x, \{(y, fib(y)) \mid y \in R(x)\}) = fib(x - 1) + fib(x - 2)$,
- so we obtain (in the specification language) ,
  $f(x) :=$
  $\mathit{if}\ x = 0 \vee x = 1\ \mathit{then}\ 1\ \mathit{else}\ fib(x - 1) + fib(x - 2)\ \mathit{fi} : \mathbb{N} \Rightarrow \mathbb{N}$

The next example is the construction of a function f that assigns to each node the length of a *shortest path in a graph* to some specific node b. The steps are as follows:

- the signature of f is $S \Rightarrow \mathbb{Q}$, where S is some type that contains all the *nodes*,
- $B = \{b\}$,
- $f(b) = 0$,
- R is a function that assigns to each node a set of nodes, so its signature is $S \Rightarrow \mathbb{F}(S)$; in fact R determines the *edges* of the graph,
- $h(x, \{(y, f(y)) \mid y \in R(x)\}) = min((y : R(x) \mid d(x, y) + f(y)))$ where d is the distance function with signature $S \times S \Rightarrow \mathbb{Q}$ and where min is a function with signature $\mathbb{F}(\$ \times \mathbb{Q}) \Rightarrow \mathbb{Q}$ that determines the minimum of the range of a binary relation.

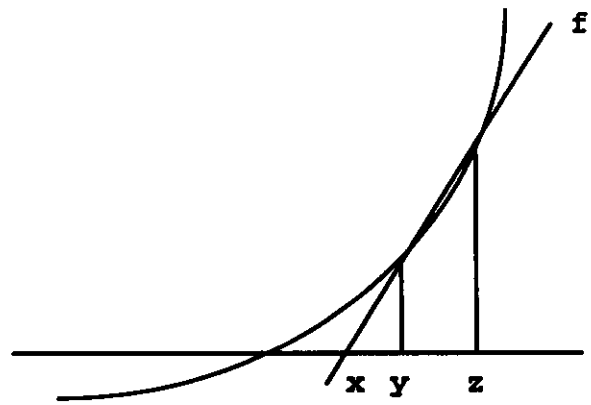

Figure 24.1: Newton-Raphson

- so we obtain in the specification language: $f(x) :=$
  $if\ x = b\ then\ 0\ else\ min((y : R(x) \mid d(x,y) + f(y))) : S \Rightarrow \mathcal{Q}$.

Note that we need to verify several conditions to guarantee that this definition is correct, such as: for each node there is a finite path to b and the range of d contains only non-negative values. A proof that iterative application ends, is required. This example is a typical case of dynamic programming.

Ackermann function

The method does not work always. Consider for instance the *Ackermann function*, defined by:

$$\begin{array}{lll} A(x,y) := & \textit{if } x = 0 \textit{ then } y+1 \\ & \textit{elseif } y = 0 \textit{ then } A(x-1,1) \\ & \qquad \textit{else} \quad A(x-1,A(x,y-1)) \\ & \textit{fi} \\ \textit{fi} & \end{array}$$

Here we see that $R(x,y)$ includes $(x-1,1)$ but also $(x-1,A(x,y-1))$ which is dependent of A. However this function is a pathological case.

Our last example is a classical problem of numerical analysis. We are looking for a root of an equation of the form $f(x) = 0$ where f is a given function with signature $\mathcal{Q} \Rightarrow \mathcal{Q}$. In fact we are already satisfied with finding an approximation for the root, i.e. a value x such that $f(x)$ is close to 0. We use the well-known *Newton-Raphson method* to solve the problem. Based on two domain values y and z that can be considered as successive approximations for the unknown x, we can derive a better approximation x using the equation

Newton-Raphson method

$$\frac{f(z) - f(y)}{z - y} = \frac{f(y)}{y - x}.$$

See figure 24.1. We follow the steps again for the function *root* that will determine an approximation for the root:

- the signature of *root* is $\mathcal{Q} \times \mathcal{Q} \Rightarrow \mathcal{Q}$,

- B is determined by the absolute value of the function value of the approximation: $B = \{(y,z) \mid z = abs(f(y)) \ \wedge \ z < \epsilon\}$, where ϵ is some given non-negative number and *abs* gives the absolute value of its arguments,
- a is defined by: $a(y,z) = y$, the last computed approximation of the root,
- R is given by $R(y,z) = \{(y - (z-y) \times \frac{f(y)}{f(z)-f(y)}, y)\}$,
- h is simple: $h(y,z) = (y - (z-y) \times \frac{f(y)}{f(z)-f(y)}, y)$,
- the solution, in the specification language, is: $root(y,z) :=$

$$\textit{if } abs(y-z) < \epsilon \textit{ then } y \textit{ else } root(y - (z-y) \times \frac{f(y)}{f(z)-f(y)}, y) \textit{ fi}.$$

A proof of the correctness of this construction requires the verification of some conditions for f (the Lipschitz condition for instance), this is however out of the scope of this book.

We call functions for which the set $R(x)$ never contains more than one element *linear recursive functions*. Most examples we have seen, belong to this class. In fact they reduce to the special case we have considered before, i.e. they are of the form:

$$f(x) := \textit{if } b(x) \textit{ then } a(x) \textit{ else } h(x, f(g(x))) \textit{ fi}.$$

Here $g(x)$ is the unique element of $R(x)$. An important subclass of the linear recursive functions is the class of *tail recursive functions*. They are characterized by the fact that:

$$h(x, f(g(x))) = f(g(x)).$$

Next we consider the problem of transforming constructions into easier ones. Tail recursive functions are important because they can be computed relatively fast by *repetition* in stead of iterated application. With "repetition" we mean the loop construction of imperative programming languages. If we have a tail recursive function construction of the form:

$$f(x) := \textit{if } b(x) \textit{ then } a(x) \textit{ else } f(g(x)) \textit{ fi}$$

then the following imperative program will compute f:

$$\textit{while } \neg b(x) \textit{ do } x \leftarrow g(x) \textit{ od } ; x \leftarrow a(x)$$

If the precondition of this repetition is $x = x_0$ then the postcondition is $x = f(x_0)$. (Note that $\leftarrow$ denotes the assignment statement and ";" the composition operator.) In general it is not possible to transform a linear recursive function construction into tail recursion, however if h has some special properties it is possible. The next theorem gives sufficient conditions. (We consider the type information afterwards.)

Theorem 24.7 Let a linear recursive construct of the form

$$f(x) := \textit{if } b(x) \textit{ then } a(x) \textit{ else } h(k(x), f(g(x))) \textit{ fi}$$

be given. Let h satisfy:

- let b, a, h, k and g be strict functions,
- there is a *unit* element e such that $\forall y : h(e, y) = y$,
- h is *associative*, i.e. $\forall x, y, z : h(x, h(y, z)) = h(h(x, y), z)$.

Then we have

$$\forall x : f(x) = r(e, x),$$

where r is defined by:

$$r(y, x) := \textit{if } b(x) \textit{ then } h(y, a(x)) \textit{ else } r(h(y, k(x)), g(x)) \textit{ fi}.$$

(Note that r is a tail recursive function.)

Proof. We show by induction that for all relevant x and y:

$$r_n(y, x) = h(y, f_n(x)),$$

where the index n refers to the n-th approximation according to the method of successive approximations. Clearly for $n = 0$ the equation holds because both sides are $\perp$. Assume the equality holds for n. Consider $r_{n+1}(y, x)$. In case $b(x) = \perp$ the assertion holds, so there remain two cases: either $b(x) = \textit{true}$ or $b(x) = \textit{false}$. In the first case we have $r_{n+1}((y, x)) = h(y, a(x))$ and also $f_{n+1}(x) = a(x)$, hence the equation holds. In the second case we have, by the induction hypothesis:

$$r_{n+1}(y, x) = r_n(h(y, k(x)), g(x)) = h(h(y, k(x)), f_n(g(x)))$$

On the other hand

$$f_{n+1}(x) = h(k(x), f_n(g(x)))$$

and therefore

$$h(y, f_{n+1}(x)) = h(y, h(k(x), f_n(g(x))))$$

The associativity of h gives the assertion.
□

Note that notwithstanding the function k in the construction, this is an example of the special case we considered before. Here we used k because we had to decompose h a bit. The types involved are as follows:

- f, a, $k : \$_1 \Rightarrow \$_2$,
- $b : \$_1 \Rightarrow \mathbb{B}$,
- $g : \$_1 \Rightarrow \$_1$,

- $h : \$_2 \times \$_2 \Rightarrow \$_2$,
- $r : \$_2 \times \$_1 \Rightarrow \$_2$.

The term "associativity" becomes more clear if the function h is represented in infix notation. Finally note that this transformation can be carried out automatically.

We illustrate this transformation with the *factorial function*, defined by

factorial function

$$fac(x) := \textit{if } x = 0 \textit{ then } 1 \textit{ else } x \times fac(x-1) \textit{ fi} : \mathbb{Q} \Rightarrow \mathbb{Q}$$

The tail recursive equivalent is:

$$r(y,x) := \textit{if } x = 0 \textit{ then } y \textit{ else } r(y \times x, x-1) \textit{ fi} : \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q}$$

Note that $h(x,y) = x \times y$, $k(x) = x$ and that $g(x) = x - 1$ in this case. Clearly h is associative and 1 is the unit element.

Transformation to tail recursion is not restricted to linear recursive constructions. Consider for example the following solution for the Fibonacci sequence:

$$\begin{array}{l} Fib(n,x,y) := \\ \quad \textit{if } n = 1 \textit{ then } y \\ \quad \textit{else if } n = 2 \textit{ then } x \\ \quad\quad \textit{else} Fib(n-1, x+y, x) \textit{ fi} \\ \quad \textit{fi} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \end{array}$$

This function is easy to transform into an imperative program with one repetition.

The last problem we consider is the transformation of a function declaration into a construction. There are very little methods to do this. It is more an art than a trade and it requires often background knowledge, for instance in the form of theorems. Consider for instance a specification of the function *root* that is constructed above:

$$root :: \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q} :: \forall x : \mathbb{Q} \bullet \forall y : \mathbb{Q} \bullet$$

$$root(x,y) = x \Leftrightarrow abs(f(x)) < \epsilon.$$

There is no way to derive the construction given above from this specification. Sometimes a construction is the best specification we can give. Consider for instance the function fac. Informally we would specify this as:

$$fac(x) = 1 \times 2 \times \ldots \times x.$$

However if we try to formalize this we will note that
$fac(x) = x \times fac(x-1)$ in case $x \neq 0$, which is almost the construction.

Next we consider the specification of the function *union* again.

$$union :: \mathbb{F}(\$) \times \mathbb{F}(\$) \Rightarrow \mathbb{F}(\$) :: \forall x : \mathbb{F}(\$) \bullet \forall y : \mathbb{F}(\$) \bullet$$

$$\forall z : \$ \bullet z \in union(x,y) \Leftrightarrow z \in x \vee z \in y.$$

To derive a construction from this specification we have to infer that for x and y of type $\mathbb{F}(\$)$:

1. *union* is associative, i.e. $union(union(x, y), z) = union(x, union(y, z))$,
2. $union(\emptyset, y) = y$,
3. $union(\{a\}, y) = ins(a, y)$,
4. $ins(pick(x), rest(x)) = x$.

The first two properties are easy to prove, while the next two require some knowledge of the primitive functions *ins*, *pick* and *rest*. Now we are able to rewrite $union(x, y)$, in case $x \neq \emptyset$ as follows:

$$\begin{aligned} union(x, y) &= union(ins(pick(x), rest(x)), y) \\ &= union(union(\{pick(x)\}, rest(x)), y) \\ &= union(\{pick(x)\}, union(rest(x), y)) \\ &= ins(pick(x), union(rest(x), y)). \end{aligned}$$

Here we used the properties in the following order: 4, 3, 1 and 3. From this we derive the well-known construction:

$$union(x, y) := \textit{if } x = \{\} \textit{ then } y \textit{ else } ins(pick(x), union(rest(x), y))$$

(we left out the signature; it was given above.) This example illustrates that *term rewriting* is a good technique to derive properties that can be used in a function construction.

Chapter 25

Specification methods

The specification of an actor model requires that all the complex classes should be mapped to a value type and that for each processor a schema is defined. In this chapter we give some methods for finding of a suitable value type for complex classes for a schema for a processor.

25.1 Value types for complex classes

Remember that sometimes a complex class is rather trivial. If for instance, the complex class contains only one simplex class and satisfies the (only possible) tree constraint. In that case the complex class will have the type of the simplex class. For simplex classes we are free to choose a value type and in many cases we define a basic type for them, sometimes one of the standard basic types like $\mathbb{N}$, $\mathbb{Q}$ or $\mathbb{B}$ and sometimes some new basic types. In the latter case we could also have some new primitive functions. However in most cases we do not introduce new primitive functions for new basic types, which means that we can only compare two values by means of the equality function (=) and that we may perform set operations on them.

There are various ways to represent a complex class by a value type. The constraints and the functions we will apply to the complexes influence our choice. There is however one *standard construction* to represent a complex class by a *schema*. This construction is studied first and afterwards we will exploit the constraints to obtain representations that are easier to use in functions and processor relations.

Consider a complex class with name c with simplex classes with names $s_1, \ldots, s_n$ and relationships with names $r_1, \ldots, r_m$. Assume that the simplex classes are given by (defined) types with names $S_1, \ldots, S_n$ respectively. Then the complex class c is represented by the schema:

$$\begin{array}{l}[\ s_1 : \mathbb{F}(S_1), \ldots, s_n : \mathbb{F}(S_n), \\ \quad r_1 : \mathbb{F}(S_{DM(r_1)} \times S_{RG(r_n)}), \ldots, r_m : \mathbb{F}(S_{DM(r_m)} \times S_{RG(r_m)}) \mid \\ \quad r_1 \subset prod(s_{DM(r_1)}, s_{RG(r_1)}), \ldots, r_m \subset prod(s_{DM(r_m)}, s_{RG(r_m)})].\end{array}$$

Note that the subscripts DM and RG of the variables do not belong to the language, so this is a mixture of meta language and specification

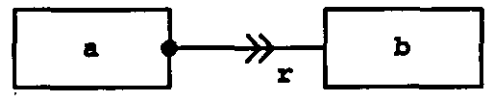

Figure 25.1: A simple complex class

language. In this definition we used the function *prod* that assigns to two sets their Cartesian product. Note that we have a type constructor for forming of the Cartesian product of types but no function yet to do the same for sets. The construction of *prod* can be found in the Toolkit.

It is easy to verify that this value type is a correct representation of the function *com* defined in part II. If we have other constraints we can add them to the predicate part of the schema. In chapter 13 we used the specification language already to express constraints. There we considered every simplex class as a basic type like we do here. Consider the simple complex class C displayed in figure 25.1. Then we have the following schema definition for C:

$$C := [a : I\!F(A), b : I\!F(B), r : I\!F(A \times B) \mid r \subset prod(a, b)].$$

For each relationship r we can define functions like we did in chapter 13. Now we define:

$$r(x) := setapply(r, x) : A \Rightarrow I\!F(B).$$

(Note that we overload the name r.) If we want to express a cardinality constraint for r, for instance that r is functional and surjective, then we add to the predicate of the schema C:

$$\forall x : A \bullet x \in a \Rightarrow size(r(x)) = 1.$$

This is a non-executable expression! However we can always transform this into an executable one since the domain of quantification is finite.The executable form of this constraint is (with function *forall* defined in toolkit):

$$forall((x : a \mid size(r_1(x)) = 1).$$

There is no need for an executable form of a constraint in case we can *prove* that the constraint is invariant for all transitions of the actor. However if we cannot prove this, then we use the constraint as part of a *postcondition* in a processor relation, and then the executability might be essential. (Note that then the invariance of the constraint is fulfilled in a trivial way, namely by allowing only transitions that keep the constraint valid.) In most cases we will not *test* a constraint in a processor relation completely and so there is no need for executability.

There is one important case where testing of a constraint is necessary, and that is if tokens from an outside source enter the system and that it is not guaranteed that these tokens are correct. Then we may use

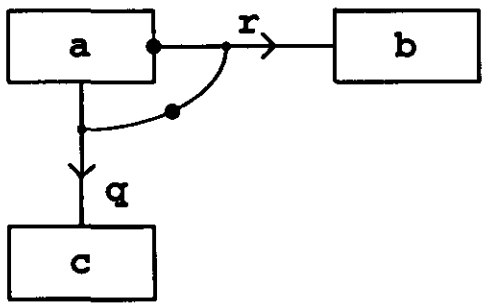

Figure 25.2: Exploitation of a key constraint

the constraint as a *precondition* in a processor relation. Note that the standard constraints can be transformed into predicates or executable expressions automatically. Since we now have a standard construction for all complex classes, we are able to express for all characteristic modeling problems of chapter 13 a suitable value type. However sometimes the constraints allow us to find a more convenient value type.

There are several constraints that can be exploited to obtain simpler representations than the standard type for a complex class. In all these cases it is easy to find the one-one transformations that map the instances of the schemas to the corresponding complexes.

First we consider the case where we have a relationship that is *total* and *surjective*. In this case we do not have to represent the simplex classes separately in the schema. Suppose in the example of figure 25.1, we have that relationship r is total and surjective. Then the schema can be reduced to a schema without predicate (i.e. a tuple type):

$$C := [r : I\!F(A \times B)]$$

because the simplexes in a complex of class C can be derived from r. So if we want to refer within the schema's predicate to the simplexes of class A and B then we use dom(r) and rng(r), respectively inside a schema with r as attribute.

The second case we consider is a *domain key constraint* formed by *total* and *functional* relationships. Consider the complex class D displayed in figure 25.2. Since relationships r and q form a key, we can define the complex class D by:

domain key constraint

$$D := [a : I\!F(B \times C), b : I\!F(B), c : I\!F(C) \mid a \subset prod(b, c)].$$

The relationships are here implicit.

The next case concerns a *tree constraint* with some *total, functional and surjective* relationships. (Note that this kind of structure often occurs.) In figure 25.3, we display complex class E and A as root simplex class. A schema definition is:

tree constraint

$$E := [a : A, c : C, d : D, b : I\!F(B)].$$

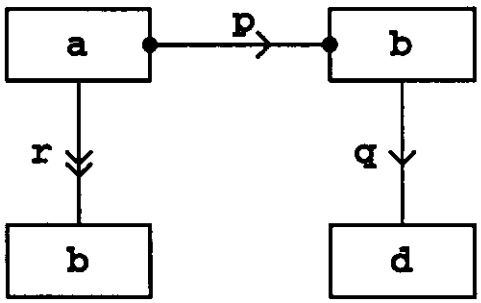

Figure 25.3: Exploitation of tree constraint

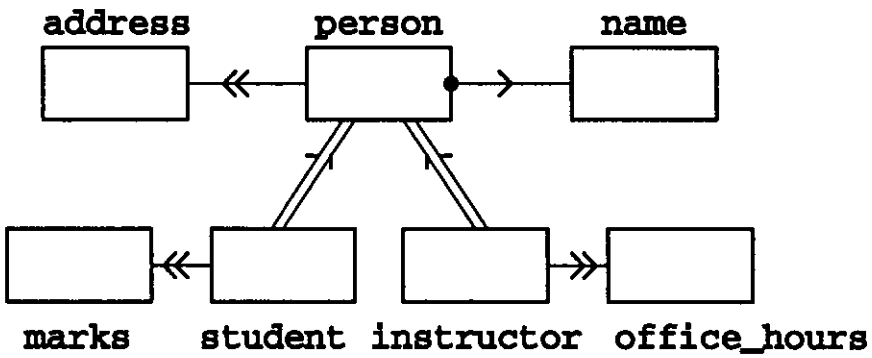

Figure 25.4: Exploitation of inheritance

(Note that this just a tuple type.) So the relationships are all implicit here and again we do not have to worry about the constraints. In case there were two or more relationships between the simplex classes, the schema definition would have the relationship names as attributes as well.

inheritance constraint
exclusion constraint

In the final case we consider *inheritance constraints* in combination with *exclusion constraints* . In figure figure 25.4 we display a complex class S as an example. In addition to the mentioned constraints we assume that a tree constraint holds with *Person* as root. A schema definition for S is:

$$\begin{aligned} S := [\ & p : Person, a : I\!F(Address), n : Name, m : I\!F(Marks), \\ & o : I\!F(OfficeHours), k : Kind \mid \\ & k \in \{'student', 'instructor'\} \ \wedge \ k =' student' \Rightarrow o = \{\} \ \wedge \\ & k =' instructor' \Rightarrow m = \{\}]. \end{aligned}$$

So we did not represent the simplex classes *Student* and *Instructor* directly but we used another attribute to make the distinction between the two kinds of persons. This "trick" is on the level of object modeling not recommended because it would introduce constraints that involve specific simplexes, however on this level of specification it might be handy, because it gives a simple schema definition.

We conclude this section with the representation of the *relational data model* in the specification language. In chapter 13 we have seen how we can represent our object model in several other frameworks, for instance the relational data model. These transformations where useful in case the systems engineer wants to continue the specification process in another framework. However the type system of our specification language has schemas which can be considered as a generalization of the relations of the relational data model. Therefore it should be easy to express a relational model into schemas. If we combine this transformation with the transformation from an object model into a relational model then we have another "standard" type for complex classes (note that we restrict us to one complex class that might be considered as a universal complex class). Consider again the relational schema studied in section 13.3:

relation	attribute	domain	key
r_1	a_1	A_1	n
	a_2	A_1	y
	a_3	A_2	y
r_2	a_4	A_2	y
	a_5	A_3	y
	a_6	A_3	n
r_3	a_7	A_3	y
	a_8	A_4	y
	a_9	A_4	n

(Note the difference between a "relational schema" and a "schema"in the sense of the language.) A schema for this relational model is defined in two steps: first we define tuple types for each table and afterwards we define a schema D for the whole database. Note that we have to take care of the key constraints.

$$
\begin{aligned}
R_1 &:= [\ a_1 : A_1, a_2 : A_1, a_3 : A_2] \\
R_2 &:= [\ a_4 : A_2, a_5 : A_3, a_6 : A_3] \\
R_3 &:= [\ a_7 : A_3, a_8 : A_4, a_9 : A_4] \\
D &:= [\ r_1 : \mathbb{F}(R_1), r_2 : \mathbb{F}(R_2), r_3 : \mathbb{F}(R_3) \mid \\
&\quad \forall x : R_1,\ y : R_1 \bullet x \in r_1 \ \wedge\ y \in r_1 \ \wedge \\
&\quad \pi_{a_2}(x) = \pi_{a_2}(y) \ \wedge\ \pi_{a_3}(x) = \pi_{a_3}(y) \Rightarrow x = y \ \wedge \\
&\quad \forall x : R_2,\ y : R_2 \bullet x \in r_2 \ \wedge\ y \in r_2 \ \wedge \\
&\quad \pi_{a_4}(x) = \pi_{a_4}(y) \ \wedge\ \pi_{a_5} = \pi_{a_5} \Rightarrow x = y \ \wedge \\
&\quad \forall x : R_3,\ y : R_3 \bullet x \in r_3 \ \wedge\ y \in r_3 \ \wedge \\
&\quad \pi_{a_7}(x) = \pi_{a_7}(y) \ \wedge\ \pi_{a_8}(x) = \pi_{a_8}(y) \Rightarrow x = y]
\end{aligned}
$$

For other data models similar representations can be found. In particular we can express the nested relational model directly in the specification language. (This is an exercise.)

Queries for a relational model can be expressed in the *relational algebra*. The relational algebra has the following operators: *projection*, *selection*, *rename*, *join*, *union* and *set difference*.

We show here how these operators can be "simulated" in the specification language. Consider a relation r that belongs to a tuple type with at least a and b as attributes, so

$$r : I\!F([a : \$_1, b : \$_2] \bowtie \$_3).$$

The *projection* should be defined for each attribute list. For example the projection on attribute a is:

$$\begin{array}{l} Pa(r) := \\ \quad \textit{if } r = \{\} \textit{ then } \{\} \\ \quad \textit{else } ins(\pi_a(pick(r)), Pa(rest(r))) \\ \quad \textit{fi} : I\!F([a : \$_1,\ b : \$_2] \bowtie \$_3). \end{array}$$

The *selection* selects tuples with certain values, for example the selection on attribute a that should have value x (x is a variable):

$$\begin{array}{l} Sel(r, x) := \\ \quad \textit{if } r = \{\} \textit{ then } \{\} \\ \quad \textit{elseif } \pi_a(pick(r)) = x \textit{ then } ins(pick(r), Sel(rest(r), x)) \\ \qquad \textit{else } Sel(rest(r), x) \textit{ fi} \\ \quad \textit{fi} : I\!F(([a : \$_1,\ b : \$_2] \bowtie \$_3) \times \$_1) \Rightarrow I\!F([a : \$_1,\ b : \$_2] \bowtie \$_3). \end{array}$$

The *rename* only changes attribute names. For example:

$$\begin{array}{l} Rac(r) := \\ \quad \textit{if } r = \{\} \textit{ then } \{\} \\ \quad \textit{else } ins(\{a \mapsto \pi_a(pick(r)), c \mapsto \pi_b(pick(r))\}, Rac(rest(r))) \\ \quad \textit{fi} : I\!F([a : \$_1,\ b : \$_2]) \Rightarrow\ I\!F([a : \$_1,\ c : \$_2]). \end{array}$$

These functions are specific, i.e. they have to be defined per query. The other operators are *generic* functions i.e. they are defined for arbitrary relations.

For the *join* we need an auxiliary function: *semijoin*:

$$\begin{array}{l} semijoin(x, y) := \\ \qquad \textit{if } x \oplus y = y \oplus x \textit{ then } ins(x \oplus y, semijoin(x, rest(y))) \\ \qquad \textit{else } semijoin(x, rest(y)) \textit{fi} : \\ \qquad \$_1 \times I\!F(\$_2) \Rightarrow I\!F(\$_1 \bowtie \$_2). \end{array}$$

The *join* is defined by:

$$\begin{array}{l} join(x, y) := \\ \quad \textit{if } x = \{\} \textit{ then } \{\} \\ \quad \textit{else } ins(semijoin(pick(x), y), join(rest(x), y)) \\ \quad \textit{fi} : I\!F(\$_1 \times \$_2) \Rightarrow I\!F(\$_1 \bowtie \$_2). \end{array}$$

The *union* is already defined.
The *set difference* is defined by:

$$setdif(x, y) := \{z : x | \neg(z \in y)\}.$$

So the relational algebra is incorporated in the specification language.

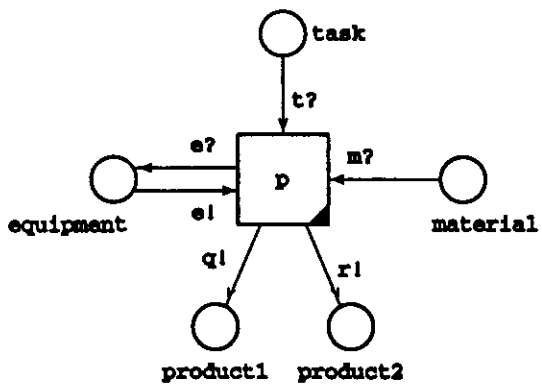

Figure 25.5: Production system

25.2 Specification of processors

The final piece of the puzzle is the specification of a processor. We use schemas to specify them. We will address the following problems:

- how does a schema defines a processor relation?
- how to deal with token identity and time stamps?
- how can we use the processor characteristics?
- how do we deal with pre and postconditions?

We will answer these questions using a simple example of a processor. In figure figure 25.5 we display this processor, that is executing tasks. A task defines a product. There are four kinds of tasks and so there are four kinds of products (1, 2, 3 and 4). Two kinds of tasks (1 and 2) require equipment, while all tasks require materials. There are two kinds of equipment (A and B) and there are also two kinds of materials (C and D). The four kinds of products are to be send to two different places: products 1 and 3 are sent to one place and the other products to the other place. In the table below we give an informal description of the processor relation:

$t?$	$e?$	$m?$	$e!$	$q!$	$r!$
1	$\perp$	C	$\perp$	1	$\perp$
2	$\perp$	D	$\perp$	$\perp$	2
3	A	C	A	3	$\perp$
4	B	D	B	$\perp$	4

We will give a schema for this processor. According to the definition of the actor model a processor relation R_p for processor p is a set of functions with domains that are subsets of the set of input and output connectors. The function values are triples consisting of an *identity*, a *value* and a *time stamp*. We will represent these triples as different

variables in the schema. (We sometimes mix up the terms "connector" and "variable" of schema that represents the processor relation.) For each connector c we have in principle three variables in the schema that represents a processor relation: c, c_i and c_t. The first one denotes the *value* of the token that is consumed or produced via connector c, the second one denotes the *identity* of the token and the last one its *time stamp*. We also use the decorations to distinguish the input and output connectors, ? and ! respectively for channels and $'$ for output to a store.

If some variable equals $\perp$ for some tuple that belongs to the schema, it denotes that the corresponding connector, i.e. the connector with the same name, no token passes during the firing. So here we give the symbol $\perp$ a specific interpretation. This interpretation fully agrees with the fact that no token should pass a connector if the connector does not appear in the domain of the firing rule. Recall that if the predicate of a schema evaluates to $\perp$, then the tuple does not belong to the schema and therefore not to the processor relation (cf. definition 23.8). Consider the following example (in which we do not consider identities and time stamps):

$$[a? : I\!N, b? : I\!N, c! : I\!N \mid a? \leq 5 \ \wedge \ c! = 2 \times a].$$

So $b?$ is free: it may be either $\perp$ or some natural number. This is a form of non-determinism we seldom want, because it is not determined if the processor will consume a token from connector $b?$ or not. If we want to exclude this, we have to add for instance, $b? \neq \perp$ in the predicate of the schema definition. Suppose now that the predicate would be extended by a conjunct:

$$b? = \frac{60}{a?}.$$

In this case it would also be unclear if a token via connector $b?$ should be consumed or not, because if $a? = 0$ then $b? = \perp$ and in all other cases ($a?$ is 1, 2, 3, 4 or 5) $b?$ is properly defined.

In general it is undecidable if a token will be consumed or produced for a connector or not, because it depends on the evaluation of an arbitrary function. An example that shows the role of $\perp$ in a schema is modification of the example above:

$$[a? : I\!N, b? : I\!N, c! : I\!N \mid a? = \perp \ \wedge \ b? = \perp \ \wedge \ c! = \perp].$$

This schema denotes a set of exactly one tuple:

$$\{a \mapsto \perp, b \mapsto \perp, c \mapsto \perp\},$$

however this tuple means that there is neither consumption nor production of tokens, so it is an incorrect definition of a processor relation, because the domain of the (only) firing rule in the processor relation is empty!

Next we consider the time stamps and identities in the processor relation and we will give a schema for the example of figure 25.5. For

the variables representing identity and time stamp of a token, we have the following types ID and $TIME$, where $ID = \mathbb{N}^*$ and $TIME = \mathbb{Q}$. (Remember that we may use sequences of natural numbers as representations for token identities and that the parent function F assigns to an identity its parent by deleting the last element of the identity.) We do not allow that the identity and time variable are defined in case the corresponding, value variable is undefined, i.e. $\perp$. The schema definition for the example of figure figure 25.5 is:

P
$t? : \mathbb{N}$
$e? : CHAR$
$m? : CHAR$
$e! : CHAR$
$q! : \mathbb{N}$
$r! : \mathbb{N}$
$t_i?, e_i?, e_i!, m_i?, q_i!, r_i! : ID$
$t_t?, e_t?, e_t!, m_t?, q_i!, r_i! : TIME$
if $t? = 1 \wedge e? = \perp \wedge m? =$ "C" *then* $e! = r! = \perp \wedge q! = 1$
else
if $t? = 2 \wedge e? = \perp \wedge m? =$ "D" *then* $e! = q! = \perp \wedge r! = 2$
else
if $t? = 3 \wedge e? =$ "A" $\wedge m? =$ "C" *then* $e! =$ "A" $\wedge q! = 3 \wedge r! = \perp$
else
if $t? = 4 \wedge e? =$ "B" $\wedge m? =$ "D" *then* $e! =$ "B" $\wedge q! = \perp \wedge r! = 4$
else true fi fi fi fi
$h := max(\{t_t?, e_t?, m_t?\}) + f(t?) : \mathbb{Q}$
if $e! \neq \perp$ *then* $F(e_i!) = t_i? \wedge e_t! = h$ *else* $e_i! = e_t! = \perp$ *fi*
if $q! \neq \perp$ *then* $F(q_i!) = t_i? \wedge q_t! = h$ *else* $q_i! = q_t! = \perp$ *fi*
if $r! \neq \perp$ *then* $F(r_i!) = t_i? \wedge r_t! = h$ *else* $r_i! = r_t! = \perp$ *fi*
$q_i! \neq \perp \Rightarrow q_i! \neq e_i!$
$r_i! \neq \perp \Rightarrow r_i! \neq e_i!$

The function f defines for task $t?$ the production time. The type $CHAR$ is a basic type of characters. The function max should be defined in a way that it ignores $\perp$.

The example of figure figure 25.5 shows that it is cumbersome to specify the time stamps and identities in this way, particular if we do not use the identities or the time stamps. Therefore we recommend to divide the specification into two schemas: one dealing with the values only and one schema for the identities and the time stamps. In case the identities or time stamps play no role, then the last schema can be generated automatically. In our example we would have a schema P_{val} and a schema $P_{id-time}$ and the total schema becomes:

$$P := P_{val} \wedge P_{id-time}.$$

Schema P_{val} has six variables, namely only the variables that deal with values and the first four predicates. Schema $P_{id-time}$ has 16 variables:

the variables that deal with time and identity and the value variables that are used to determine them, further it has the last five predicates. We call $P_{id-time}$ the *auxiliary* schema of P_{val}, which is called the *main* schema. Schema P_{var} becomes:

P_{val}
$t? : \mathbb{N}$ $e? : CHAR$ $m? : CHAR$ $e! : CHAR$ $q! : \mathbb{N}$ $r! : \mathbb{N}$
if $t? = 1 \wedge e? = \perp \wedge m? =$ "C" *then* $e! = r! = \perp \wedge q! = 1$ *else* *if* $t? = 2 \wedge e? = \perp \wedge m? =$ "D" *then* $e! = q! = \perp \wedge r! = 2$ *else* *if* $t? = 3 \wedge e? =$ "A" $\wedge m? =$ "C" *then* $e! =$ "A" $\wedge q! = 3 \wedge r! = \perp$ *else* *if* $t? = 4 \wedge e? =$ "B" $\wedge m? =$ "D" *then* $e! =$ "B" $\wedge q! = \perp \wedge r! = 4$ *else true fi fi fi fi*

As we have seen in this example, in many cases we determine the time stamp of output tokens by means of a *delay* with respect to the transition time. The transition time is always available as the maximum of the time stamps of the consumed tokens. Instead of specifying the time stamps of the produced tokens in the auxiliary schema, we write in the main schema an expression of the form:

$$x_t! = TransTime + delay,$$

where $x!$ is an output variable, $TransTime$ is a value equal to the maximum of the time stamps of the consumed tokens and *delay* is a term that evaluates to a non-negative element of $\mathbb{Q}$. So we introduce underhand a *time variable* $TransTime$ in the main schema. It is easy to transform such a "polluted" main schema into a correct one by transferring this predicate to the auxiliary schema in the right form.

The choice of the input token that was used for identification of the output tokens, was rather arbitrary in the example above. The only thing that counts is that there is really an input token for the connector.

There are cases in which it is important to use the identities of tokens also in *values* of tokens. Sometimes a simplex in the complex of a token represents an identity, for example an order number or a transaction identity. In these cases it is very convenient that we have an always available source of new identities, namely the token identities. In such a case we may give an output variable $y!$ an identity as value provided that it has ID as type. Formally this requires a quite complex predicate that includes $F(y!) = x_i?$ as a conjunct, in which $x_i?$ is the input token that is used for the generation of new identities. The rest of the predicate

states that all other children of $x_i?$, either used as identity of an output token or as a value, should be different. It is not difficult to express this, but it can be generated automatically. Therefore we introduce another keyword, like *TransTime* above, namely *New* and we use it like:

$$y! = New.$$

A specification with these two keywords in it can be transformed into a "correct" one.

If we use identities as values then we can do more than just comparing them by means of $=$ and $\neq$. It is sometimes interesting to check if one identity is a prefix of another or if they have a common prefix. These questions arise in *object oriented modeling*.

Next we consider the *processor characteristics*. The processor in the example above is neither input nor output *complete*, which is easy to verify by the occurrence of $\perp$ in the schema. The processor is not *total* either, since there are several combinations of values of input tokens for which there is no firing rule. However the processor if *functional*. Functionality also includes that if a processor can fire with for example n specific input tokens, then it cannot fire with more tokens including these n. For a complete and functional processor we can find a function that determines the processor relation. Consider for example a processor with input connectors $a?$, $b?$ and $c?$ and output connectors $x!$ and $y!$. Then there should be a function f such that:

$$x! = \pi_1(f(a?, b?, c?)) \ \wedge \ y! = \pi_2(f(a?, b?, c?)).$$

In general the predicate in the main schema of a processor relation has the following format:
if $p_1(x_1?, \ldots, x_m?)$ *then* $q_1(x_1?, \ldots, x_m?, y_1!, \ldots, y_n!)$ *else*

.....

if $p_k(x_1?, \ldots, x_m?)$ *then* $q_k(x_1?, \ldots, x_m?, y_1!, \ldots, y_n!)$ *else false*
fi ...*fi*.
Here the Boolean functions $p_1, \ldots, p_k$ are the preconditions and the Boolean functions $q_1, \ldots, q_k$ are the postconditions.

If at least one precondition evaluates to *true* for an input variable equal to $\perp$, then the processor is not input complete. In case the processor relation is functional, the postconditions can be transformed into the following form:
$q_i(x_1?, \ldots, x_m?, y_1!, \ldots, y_n!) =$

$$y_1! = f_{i,1}(x_1?, \ldots, x_m?) \ \wedge \ \ldots \ \wedge \ y_n! = f_{i,n}(x_1?, \ldots, x_m?).$$

In that case the processor specification is *executable*.

References and Further Reading

The *specification language* is very close to the language Z. The main difference is that we do not consider a function signature as a type. Further our language has a constructive (and therefore executable) subset, i.e. a *functional language*. Main literature for Z is [Spivey, 1987] for the semantics, [Spivey, 1989] for the language and the toolkit, and [Wordworth, 1992; Hayes, 1987; Woodcock and Loomes, 1988] for the specification methodology, including verification of properties. The language Z is close to the language of VDM, see [Jones, 1990; Andrews and Ince, 1991]. In the VDM literature more attention is paid to stepwise refinement and verification.

An important aspect of specification is the recursive definition of functions, which is the key issue of *functional programming*. Good books for functional programming are: [Glaser *et al.*, 1984], [Meyer, 1990] and [Wilkstrom, 1987]. A good reference for *type theory* in combination with functional programming is [Thompson, 1991]. Type systems with tuple types usually adopt Cardelli's method for polymorphy (see [Cardelli and Wegner, 1985]). We do not need this approach because of the $\bowtie$ operator for types. Functional languages are based on *lambda calculus* and the main reference is [Barendregt, 1984]. For set theory and predicate logic see [Enderton, 1977].

Exercises

1. Prove theorem 21.3.
2. Prove the remaining cases of theorem 21.5.
3. Prove the following equality in schema calculus:

$$\exists x : T \mid p \bullet [x : T, y_1 : S_1, \ldots, y_n : S_n \mid q]$$
$$=$$
$$([x : T \mid p] \wedge [x : T, y_1 : S_1, \ldots, y_n : S_n \mid q]) \backslash (x)$$

4. Give type definitions to represent the nested relational model in the specification language.
5. Prove that the function *dom* (see the toolkit) satisfies the declaration:
signature $dom : \mathbb{F}(\$_1 \times \$_2) \Rightarrow \mathbb{F}(\$_1)$
predicate $\forall f : \mathbb{F}(T \times S) \bullet \forall x : T \bullet$
$\exists y : \$_2 \bullet (x, y) \Leftrightarrow x \in dom(f)$.
6. Give a declaration and a construction for a function *update*, that assigns to a binary relation f with signature $\mathbb{F}(\$_1 \times \$_2)$ and two elements $x \in \$_1$ and $y \in \$_2$ a new binary relation r', that contains the pair (x, y) and satisfies:

$$\forall v : \$_1 \bullet \forall w : \$_2 \bullet x \neq v \Rightarrow ((v, w) \in r \Leftrightarrow (v, w) \in r')$$
$$\wedge$$
$$\forall w : \$_1 \bullet (x, w) \in r' \Rightarrow w = y.$$

Prove that the result of the function *update* is a functional binary relation if it is applied to a functional binary relation.
7. Give a declaration and a definition of the function *In* that assigns to two integers k and l the set of all integers i that satisfy: $k \leq i \leq l$.
8. Give a declaration and a construction for a function that assigns to an arbitrary singular value a suitable type expression.
(Hint: represent a singular value and a type expression as a sequence.)
9. Give declarations and definitions of two functions that compute the union and the intersection respectively, of the elements of a set of sets.
10. Give declarations and definitions for two functions that compute the mean and standard deviation of a frequency distribution, i.e. the arguments are of type $\mathbb{F}(\mathbb{Q} \times \mathbb{N})$ and have unique first components.

11. Consider the Car Rental Company of exercise 6 in part III again. The rental prices are based on the following rules. There are three types of cars: compact, midsize and fullsize. The rental price of a car is a *basic day price* (depending on the type of car) multiplied by a *daily discount (factor)* for the length of the rental period and by the number of days of the rental. Further a *drop off charge* has to be paid if the car is delivered by the client to another station of CRC. The drop off charge is equal to the basic day price, if the car is returned to a station at a distance not further than 1000 km and two times the basic day price in case it is delivered at a farther station.
There are three exceptions to this rule:

    (a) if the car is rented for a longer period (i.e. longer than 15 days) and it is not a compact car, then the drop off charge for long distance (i.e. more than 1000 km) is only one basic day price,

    (b) if a fullsize car is rented for at least 6 days, but no longer than 15 days, then the drop off charge for long distance is also only one basic day price,

    (c) if a fullsize car is rented for at least 16 days and it is returned at a station not further than 1000 km, no drop off charge has to be paid.

    There is no *daily discount* if a car is rented for less than 6 days. If a compact car is rented for a period longer than 5 and shorter than 16 days, the daily discount is 10% if the car is returned to the rental station, 5% if it is returned to another station not further than 1000 km and there is no discount in case it is returned to a station further than 1000 km.
    If a compact car is rented for a long period (longer than 15 days), then the discount is 20% if the car is returned to the rental station, 10% if it is returned to another station not further than 1000 km and 5% in case it is returned to a station further than 1000 km.
    For midsize and fullsize cars rented for a period longer than 5 days and no longer than 15 days, the discount is 20% if the car is returned to the rental station, 15% if it is returned to another station not further than 1000 km and 10% in case it is returned to a station further than 1000 km. For for fullsize cars that are rented longer than 15 days the discounts for the rental period between 6 and 16 days is increased by 15% and for midsize cars by 10%.
    The basic daily price of a compact car is $ 30, of a midsize car $ 50 and of a fullsize car $ 70.
    Give a specification for the processor that computes the rental price.

(All the functions in the toolkit can be considered as exercises as well.)

Appendix A

Mathematical notions

Mathematics and in particular mathematical logic, is used to define and analyze the frameworks and as the *meta language* for the specification language. The mathematical notations are very similar to the specification language. The basic notions used belong to *set theory*, *predicate calculus* and *lambda calculus*.

Let A, $A_1, \ldots, A_n$, B and C be *sets* and a, $a_1, \ldots, a_n$ be *elements*.

Sets

- $\emptyset$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{B}$ denote special sets: the sets of natural numbers, the integers, the rational numbers, the real numbers and the set of Boolean values (*true* and *false*), respectively. These sets are mutually disjoint.
- $a \in A$ is *true* if and only if a is an *element* of A.
- $A \subset B$ is *true* if and only if A is a *subset* of B, i.e. if and only if all elements of A are also elements of B.
- $A \cup B$ is the *union* of A and B, i.e. the set of elements that belong to A or B.
- $A \cap B$ is the *intersection* of A and B, i.e. the set of elements that belong to A and B.
- $A \backslash B$ is the *difference* of A and B, i.e. the set of all elements of A that do not belong to B.
- An *enumerated* set is denoted by $\{a_1, \ldots, a_n\}$, where $a_1, \ldots a_n$ are elements.
- $\#(A)$ is the *cardinality* of A, i.e. the number of elements of A (this number can be ∞).

Constructed sets

- $\mathbb{P}(A)$ is the *power set* of A, i.e. the set of all subsets of A.

- $\mathbb{F}(A)$ is the *finite power set* of A, i.e. the set of all finite subsets of A.

- A^* is the set of all *sequences* of elements of A, including the *empty sequence* ϵ; they are denoted by $\langle a_1, \ldots, a_n \rangle$.

- $A_1 \times \ldots \times A_n$ is called a *Cartesian product* and is the set of all *rows* $(a_1, \ldots, a_n)$ such that $a_i \in A_i$; rows of two elements are called *pairs*.

- $[b_1 : A_1, \ldots, b_n : A_n]$ is called a *tuple type* and is the set of all *tuples* of the form $\{b_1 \mapsto a_1, \ldots, b_n \mapsto a_n\}$ where $b_1, \ldots, b_n$ are different elements of a set B, they are called *attributes* in this role. For all i it should hold that $a_i \in A_i$. Formally there is no difference between a tuple and set of *pairs* $\{(b_1, a_1), \ldots, (b_n, a_n)\}$.

There are two other constructions for sets: set comprehension and the generalized Cartesian product. They are defined after the introduction of some other notions.

Functions

- A *function* is a set of pairs such that the first elements of the pairs are unique (note that a tuple is also a function).

- The *domain* of a function f is the set of first elements of the pairs that belong to the function; it is denoted by $dom(f)$.

- The *range* of a function f is the set of all second elements of the pairs that belong to the function; it is denoted by $rng(f)$.

- $A \nrightarrow B$ is the set of all functions f with $dom(f) \subset A$ and $rng(f) \subset B$; the elements of $A \nrightarrow B$ are called *partial functions*.

- $A \rightarrow B$ set of all functions $f \in A \nrightarrow B$ with $dom(f) = A$,

- Let $f \in A \nrightarrow B$ then $f \restriction C$ is the *restriction* of f to C, i.e. the set of all pairs of f such that the first element belongs to C.

- If $a \in dom(f)$ then $f(a)$ is the element in $rng(f)$ such that $(a, f(a)) \in f$; $f(a)$ is called the *application* of f to a.

- For functions on a Cartesian product applications are sometimes represented by subscripts, for example $f(a, b)$ as $f_a(b)$ and $f(a, b, c)$ as $f_{a,b}(c)$.

- If $f \in A \nrightarrow (B \nrightarrow C)$ then f is a *function-valued* function and then there is an equivalent function $\tilde{f} \in (A \times B) \nrightarrow C$ such that for all a and b: $f(a)(b) = \tilde{f}(a, b)$; $\tilde{f}$ is called the *curried version* of f.

- If F is a *set-valued* function (i.e. the range elements are sets) then $\Pi(F)$ is the *generalized Cartesian product*, i.e. the set of all functions f such that $dom(f) = dom(F)$ and for all $x \in dom(F)$: $f(x) \in F(x)$.
- The *inverse* of a function $f \in A \nrightarrow B$ is a set-valued function f^{-1} such that, for $b \in B$ $f^{-1}(b)$ is the set of all elements of $a \in A$ such that $f(a) = b$.
- A function $f \in A \nrightarrow B$ is called *injective* if for all a, b in $dom(f)$ with $a \neq b$: $f(a) \neq f(b)$.
- A function $f \in A \nrightarrow B$ is called *surjective* if $rng(f) = B$,
- A function $f \in A \rightarrowtail B$ is called *bijective* if it is injective and surjective.
- A function $f \in A \nrightarrow B$ is called *total* if $dom(f) = A$ (so $f \in A \rightarrow B$).

Predicates

- Functions $f \in A \nrightarrow \mathbb{B}$ are called *Boolean functions* or *predicates*.
- If a and b are Boolean values then $\neg a$, $a \wedge b$, $a \vee b$, $a \Rightarrow B$ are also Boolean values, denoting the *negation*, *conjunction*, *disjunction* and *implication* respectively.
- For a predicate $p \in A \nrightarrow \mathbb{B}$ the *universal* and *existential quantification* over the set A are denoted by: $\forall x \in A : p(x)$ and $\exists x \in A : p(x)$, respectively.
- If p is a predicate then $\{x \in A \mid p(x)\}$ is the set of all elements a of A for which the $p(a)$ evaluates to *true*; also the notation $\{x \mid p(x)\}$ is used which means that A has to be replaced by $dom(p)$; this notation is called *set comprehension*.

Ordering

- A *partial ordering* on a set A, denoted by a symbol $\leq$, is a predicate in $A \times A \nrightarrow \mathbb{B}$ (denoted in infix notation, i.e. we write $a \leq b$ instead of $\leq (a, b)$), such that:
  - $a \leq a$, for all $a \in A$ (*reflexivity*)
  - if $a \leq b$ and $b \leq a$ then $a = b$ (*anti-symmetry*)
  - if $a \leq b$ and $b \leq c$ then $a \leq c$ (*transitivity*).
- If $A \subset B$ then b is $sup(A)$, called the *supremum* or the *least upper bound* of A with respect to B, if for all $a \in A : a \leq b$ and there is no other element with this property in B. (It can be proved that the *sup* is unique.)

- Similarly b is $inf(A)$, called the *infimum* or the *greatest lower bound* of A with respect to B, if for all $a \in A : b \leq a$ and there is no other element with this property in B.

Lambda calculus

Lambda calculus is a formalism that is used in this book to define functions. It has its own language and rules to derive new expressions from given expressions. First we define this language:

- there is a set of *variables* and a set of *constants*; they are disjoint,
- each variable and each constant is an expression,
- if E_1 and E_2 are expressions then $E_1(E_2)$ is an expression, called the *application* of E_1 to E_2,
- if x is a variable and E an expression then $(\lambda x \bullet E)$ is an expression, called an *abstraction* or a *lambda expression*.

For expressions we have the following *rewrite rules*. Let x and y be a variables and E an expression. (We use "=" to express that two expressions can be obtained from each other by applying a rewrite rule.)

- *α-conversion*: $(\lambda x \bullet E) = (\lambda y \bullet E_y^x)$, where E_y^x denotes the expression E with each occurrence of x replaced by y. Here we assume that y does not occur (free) in E (see part V for a definition of "free occurrence" of a variable).
- *β-reduction*: $(\lambda x \bullet E)(y) = E_y^x$.
- *η-reduction*: $(\lambda x \bullet E(x)) = E$.

A rewriting step is called a *reduction* if the number of λ's has decreased. An expression that cannot be reduced is called a *normal form*. The (first) Church-Rosser theorem states that an expression can be reduced to at most one normal form. If all normal forms can be *evaluated*, i.e. have a value, then all expressions with a normal form obtain the value of their normal form. As an example consider:

$$(\lambda x \bullet f(g(x))$$

where f and g are constants that denote given functions. Then the lambda expression denotes the function $\{(a, f(g(a)) \mid a \in dom(g)\}$, if $rng(g) \subset dom(f)$. In general if x is the only variable in E then the lambda expression $(\lambda x \bullet E)$ denotes the function consisting of all pairs (a, E_a^x). If it is not clear from the context which elements a we have to consider we write: $\lambda x \in A \bullet E$, to specify that we have to consider all elements of A for which the expression E can be evaluated. This is an expression in *typed lambda calculus* because all variables have a domain or type. This is the lambda calculus we use in the specification language. The variables in an expression may be place holders for functions, for example $(\lambda y \bullet y(a))$ denotes the function that assigns to an

arbitrary function y the function value for argument a (a is a constant here). An example of reduction:
$(\lambda y \bullet (\lambda x \bullet y(x)))(a) =$ (by β-reduction)
$(\lambda x \bullet a(x)) = a$ (by η-reduction).

Principle of structural induction

Structural induction is a generalization of induction over the natural numbers. Suppose we have a finite set of rules to construct objects out of given objects and that we have a finite set of atomic objects (i.e. objects that are not constructed out of others). If we have to show that a property holds for all objects then we have to show:

- all atomic objects have the property,
- assuming that all components of an arbitrary object have the property we have to prove that the object has the property.

If we take the natural numbers as objects and the construction rule is "addition by one" then the principle of structural induction says that we have to prove the property for 0 and under the assumption that it holds for n we have to show that it holds for $n + 1$.

Appendix B

Syntax summary

Meta syntax

- The definition sign for non-terminals is ::=.
- Any part of a syntax in underlined typeface is to be taken literally.
- Any part between '{}' braces may be repeated. So 'a ::= $\{b\}$' is shorthand for 'a ::= $b \mid b\,a$'.
- Any part between "[ ]" square brackets may be omitted. So 'a ::= $[b]c$' is shorthand for 'a ::= $bc \mid c$'.
- Any part between '<>' triangular brackets may be repeated; each repetition must be preceded by a comma ','. So '$< a >$' is shorthand for '$a[\{,a\}]$'.
- The syntax for identifiers, digits and characters is not further elaborated.

Syntax base
The syntax base consists of the following sets:

$$(L, C, TV, V, VN, FN, TN, SN)$$

with:

L: the set of attributes,
C: the set of constants,
TV: the set of type variables,
V: the set of value variables,
VN: the set of value names,
FN: the set of function names,
TN: the set of type names,
SN: the set of schema names.

The set TN contains the names of the basic types, i.e. at least $\emptyset$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{B}$. The sets TV and TN are disjoint and $TV = \{\$, \$_1, \$_2, \ldots\}$. The sets V and VN are disjoint. The sets L and V satisfy L and V are not disjoint. The syntactical variables that range over these sets are:

- *constant* $\in C \cup \{\perp\}$
- *variable* $\in V$
- *attribute* $\in L$
- *a.variable* $\in V \cap L$
- *value name* $\in VN$
- *type variable* $\in TV$
- *type name* $\in TN$
- *function name* $\in FN$
- *quantor* $\in \{\forall, \exists\}$
- $\theta \in \{\vee, \wedge, \Rightarrow\}$

Type expressions and type definitions

type expression ::=
type name | *type variable* | *set type* |
product type | *sequence type* | *tuple type* |
(*type expression*)

set type ::= $\mathbb{F}$ (*type expression*)

product type ::= *type expression* $\times$ *product list*

product list ::= *type expression* | *product type*

sequence type ::= *type expression**

tuple type ::=
type variable | [(*attribute* : *type expression*)]
| *tuple type* $\bowtie$ *tuple type*

type definition ::= *type name* $:=$ *type expression*

Terms

term ::=
variable | *value name* | *constant* | (*term*) |
value construction | *application* | *set term* | *map term*

value construction ::=
set construction | *row construction* |
sequence construction | *tuple construction*

set construction ::= {(*term*)} | {}

row construction ::= ((*term*)) | ()

sequence construction ::= ⟨(*term*)⟩ | ⟨⟩

tuple construction ::= $\{\langle attribute \mapsto term \rangle\} \mid \{\}$

application ::= *function name* $(term)$

set term ::= $\{$ *value variable* : *domain* $\mid$ *term* $\}$

map term ::= $(\langle value\ variable \rangle : domain \mid term)$

domain ::= *set construction* | *set term*

value definition ::= *value name* $:=$ *term* : *type expression*

Function definition

function definition ::=
function name$(\langle variable \rangle) :=$ *term* : *signature*,

signature ::= *type expression* $\Rightarrow$ *type expression*.

Predicates

predicate ::= *bool* | $\neg$ *predicate* | $(predicate\ \theta\ predicate)$ |
quantor $\langle variable \rangle$: *domain* $\bullet$ *predicate*

domain ::= *type expression*

bool ::= *Boolean term*

A Boolean term is a term with type $\mathbb{B}$.

Function declaration

function declaration ::= *function name* :: *signature* :: *predicate*

Schema

schema ::= $[schema\ signature \mid predicate]$

schema signature ::= $\langle$ *a.variable* : *type expression* $\rangle$

Schema expression

schema expression ::=
schema | $(schema\ expression)$ |
schema name$(\langle$ *type expression* $\rangle)$ |
$\neg$ *schema expression* |
schema expression θ *schema expression* |
schema expression $\setminus (\langle a.variable \rangle)$ |
schema expression $\upharpoonright (\langle a.variable \rangle)$ |
quantor a.variable $\mid$ *predicate* $\bullet$ *schema expression*

schema definition ::=
schema name $[(\langle type\ variable \rangle)] :=$ *schema expression*

The following conditions should hold:

- the variables (a.variable) left of the symbols \ and ↾ should occur in the schema signatures of the schema expression,
- the variable behind a quantor should appear in a signature of the schema expression with the same type expression, and the predicate should have no free variables or type variables,
- the schema expression in a schema definition may contain type variables, however these type variables must appear also on the left hand side of the ":=" symbol,
- schema definitions may not be recursive.

Script

script ::= *line* | *line*; *script*

line ::= *type definition* | *value definition* |
function definition | *function declaration* |
schema definition

There should be a function, that assigns a natural number n to every line ℓ, with the property that every name occurring in the definition part of the line is defined itself in a line with a number smaller or equal to n in case the line is a function definition and in a line with a smaller number otherwise.

Appendix C

Toolkit

The functions are grouped by their kind. They are presented in the following format:

- "user" name (for example "equality"),
- symbolic name and signature (for example $= : \$_1 \times \$_2 \Rightarrow \mathbb{B}$),
- – infix or prefix; which indicates how we use the function (for example "=" is used in infix notation: $a = b$, while the prefix notation is $= (a, b)$),
  – strict or non-strict; in the latter case we have to modify the definition sometimes to guarantee this (for example if f is defined but not yet strict, then we modify its definition by

$$\tilde{f}(x) := \textit{ if } x = \bot \textit{ then } \bot \textit{ else } f(x) \textit{ fi},$$

  – primitive or derived; in the first case the function is defined in the meta language and in the latter case in the specification language,
- definition of the function without signature; if a function name is overloaded there are several definitions,
- auxiliary definitions, if necessary.

General functions

1. **equality**
   - $= : \$_1 \times \$_2 \Rightarrow \mathbb{B}$
   - infix / non-strict / primitive
   - this function compares two values and if they are identical or equivalent (in case of sets and tuples) then the function value is *true* else it is *false*.
     $\bot = \bot$ is true $\wedge$ $\forall x \in U : x \neq \bot \Rightarrow x = \bot$ is false.
     Note that if "=" is applied to values of different types it will always be *false*.

2. **selection**

- *if.then.else.fi* : $\mathbb{B} \times \$ \times \$ \Rightarrow \$$
- infix / non-strict and lazy / primitive
- *if a then b else c fi*,
  if a is *true* then the function value is equal to b else to c;
  the function is lazy: if a is *true* then c may be $\perp$ and if a is *false* then b may be $\perp$;
  if a is $\perp$ then the function value is $\perp$.

Numerical functions

3. **subtraction**

- $- := \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{Z}$
  $- := \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q}$
- infix / strict / primitive
- the meaning is the well-known subtraction

4. **integer division**

- *div* : $\mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
  div : $\mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{Z}$
- infix / strict/ primitive
- the meaning of a *div* b is the maximal number of b's contained in a, division by 0 gives $\perp$

5. **rational division**

- $\div$: $\mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q}$
- infix / strict / primitive
- this is the well-known division for rational numbers; division by 0 gives $\perp$

6. **truncation to integer**

- *truncint* : $\mathbb{Q} \Rightarrow \mathbb{Z}$
- prefix / strict / primitive
- $truncint(x) = max\{y \in \mathbb{Z} \mid y \leq x\}$

7. **truncation to natural**

- *truncnat* : $\mathbb{Z} \Rightarrow \mathbb{N}$
  truncnat : $\mathbb{Q} \Rightarrow \mathbb{N}$
- prefix / strict / primitive
- if $x \in \mathbb{Z}$ and $x \geq 0$ then: $truncnat(x) = x$,
  if $x < 0$ then $truncnat(x) = 0$,
  else $truncnat(x) = max\{y \in \mathbb{N} \mid y \leq x\}$

8. **conversion to integer**

   - $toint : \mathbb{N} \Rightarrow \mathbb{Z}$
   - prefix / strict / primitive
   - $toint(x) = +x$

9. **conversion to rational**

   - $torat : \mathbb{N} \Rightarrow \mathbb{Q}$
   - prefix / strict / primitive
   - $torat(x) = +x/1$

10. **addition**

    - $+ : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
      $+ : \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{Z}$
      $+ : \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q}$
    - infix / strict / derived
    - $x + y := truncnat(toint(x) - ((+0) - toint(y)))$, for the first function,
      $x + y := x - (0 - y)$ for the second function and
      $x + y := x - ((+0/1) - y)$ for the last function.

11. **multiplication**

    - $\times : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
      $\times : \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{Z}$
      $\times : \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{Q}$
    - infix / strict / derived
    - $x \times y := truncnat(torat(x) \div ((+1/1) \div torat(y)))$, for the first function,
      $x \times y := truncint(torat(x) \div ((+1/1) \div torat(y)))$, for the second function and
      $x \times y := x \div ((+1/1) \div y)$, for the last function

12. **modulo**

    - $mod : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
      $mod : \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{Z}$
    - infix / strict / derived
    - $x \; mod \; y := x - y \times (x \; div \; y)$

13. **power**

    - $\uparrow := \mathbb{Q} \times \mathbb{N} \Rightarrow \mathbb{Q}$
    - infix / strict / derived
    - $x \uparrow n := if \; n = 0 \; then \; (+1/1) \; else \; x \times x \uparrow (n-1) \; fi$

14. **less than**

- $<: \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
  $<: \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{B}$
  $<: \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$
- infix / strict / primitive
- these are the well-known comparison functions

15. **less or equal**

- $\leq: \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
  $\leq: \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{B}$
  $\leq: \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$
- infix / strict / derived
- $x \leq y :=$ *if* $x = y$ *then true else*
  if $x < y$ *then true else false fi*
  fi

16. **greater**

- $>: \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
  $>: \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{B}$
  $>: \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$
- infix / strict / derived
- $x > y := y < x$

17. **greater or equal**

- $\geq: \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{B}$
  $\geq: \mathbb{Z} \times \mathbb{Z} \Rightarrow \mathbb{B}$
  $\geq: \mathbb{Q} \times \mathbb{Q} \Rightarrow \mathbb{B}$
- infix / strict / derived
- $x \geq y := y \leq x$

18. **maximum**

- $max : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
- infix / strict / derived
- $x\ max\ y :=$ *if* $x \leq y$ *then* y *else* x *fi*

19. **minimum**

- $min : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
- infix / strict / derived
- $x\ min\ y :=$ *if* $x \geq y$ *then* y *else* x *fi*

20. **summation**

- $sum : \mathbb{F}(\$ \times \mathbb{Q}) \Rightarrow \mathbb{Q}$
- prefix / strict / derived

- $sum(f) := \mathit{if}\ f = \{\}\ \mathit{then}\ 0$
  $\mathit{else}\ \pi_2(pick(f)) + sum(rest(f))\ \mathit{fi}$

Boolean functions

21. **implication**
    - $\Rightarrow : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}$
    - infix / non-strict / derived
    - $x \Rightarrow y := \mathit{if}\ x = \bot\ \mathit{then}$
      $\mathit{if}\ y\ \mathit{then}\ \mathit{true}\ \mathit{else}\ \bot\ \mathit{fi}$
      else
      $\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ \mathit{true}\ \mathit{fi}$
      fi
22. **negation**
    - $\neg : \mathbb{B} \Rightarrow \mathbb{B}$
    - prefix / non-strict / derived
    - $\neg x := x \Rightarrow \mathit{false}$
23. **or**
    - $\vee : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}$
    - infix / non-strict / derived
    - $x \vee y := \neg x \Rightarrow y$
24. **and**
    - $\wedge : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}$
    - infix / non-strict / derived
    - $x \wedge y := \neg(\neg x \vee \neg y)$
25. **universal quantification**
    - $\mathit{forall} : \mathbb{F}(\$ \times \mathbb{B}) \Rightarrow \mathbb{B}$
    - prefix / strict / derived
    - $\mathit{forall}(f) := \mathit{if}\ f = \{\}\ \mathit{then}\ \mathit{true}$
      $\mathit{else}\ \pi_2(pick(f)) \wedge \mathit{forall}(rest(f))\ \mathit{fi}$
26. **existential quantification**
    - $\mathit{exists} : \mathbb{F}(\$ \times \mathbb{B}) \Rightarrow \mathbb{B}$
    - infix / strict / derived
    - $\mathit{exists}(f) := \mathit{if}\ f = \{\}\ \mathit{then}\ \mathit{false}$
      $\mathit{else}\ \pi_2(pick(f)) \vee \mathit{exists}(rest(f))\ \mathit{fi}$

Set functions

27. **insertion**

- *ins* : $\$ \times \mathbb{F}(\$) \Rightarrow \mathbb{F}(\$)$
- prefix / strict / primitive
- the function satisfies the equation: $ins(a, b) = \{a\} \cup b$

28. **choice function**

- *pick* : $\mathbb{F}(\$) \Rightarrow \$$
- prefix / strict / primitive
- the function satisfies: $pick(x) \in x$ and $pick(\{\}) = \perp$

29. **rest of set**

- *rest* : $\mathbb{F}(\$) \Rightarrow \mathbb{F}(\$)$
- prefix / strict / primitive
- the function satisfies: $rest(x) = x \backslash \{pick(x)\}$ and $rest(\{\}) = \perp$

30. **element of**

- $\in$: $\$ \times \mathbb{F}(\$) \Rightarrow \mathbb{B}$
- infix / strict / derived
- $x \in y :=$ *if* $y = \{\}$ *then false else*
  if $x = pick(y)$ *then true else*
  $x \in rest(y)$
  fi
  fi

31. **subset**

- $\subset : \mathbb{F}(\$) \times \mathbb{F}(\$) \Rightarrow \mathbb{B}$
- infix / strict / derived
- $x \subset y :=$ $forall(z : x \mid z \in y)$

32. **union**

- $\cup$: $\mathbb{F}(\$) \times \mathbb{F}(\$) \Rightarrow \mathbb{F}(\$)$
- infix / strict / derived
- $x \cup y :=$ *if* $x = \{\}$ *then* y *else* $ins(pick(x), rest(x) \cup y)$ *fi*

33. **intersection**

- $\cap := \mathbb{F}(\$) \times \mathbb{F}(\$) \Rightarrow \mathbb{F}(\$)$
- infix / strict / derived
- $x \cap y := \{z : x \mid z \in y\}$

34. **set difference**

- $\backslash : \mathbb{F}(\$) \times \mathbb{F}(\$) \Rightarrow \mathbb{F}(\$)$
- infix / strict / derived

- $x \backslash y := \{z : x \mid \neg(z \in y)\}$

35. **size**

- *size* : $\mathbb{F}(\$) \Rightarrow \mathbb{N}$
- prefix / strict / derived
- $size\,(x) :=$ *if* $x = \{\}$ *then* 0 *else* 1 + $size\,(rest\,(x))$ *fi*

Sequence functions

36. **concatenation**

- *cat* : $\$^* \times \$ \Rightarrow \$^*$
- prefix / strict / primitive
- let $a = \langle a_1, \ldots, a_m \rangle \in \$^*$ *and* $c \in \$$ then:
  $cat(a, c) = \langle a_1, \ldots, a_m, c \rangle$

37. **head of the row**

- *head* : $\$^* \Rightarrow \$$
- prefix / strict / primitive
- let $a = \langle a_1, \ldots, a_m \rangle \in \$^*$ then:
  $head(a) = a_1$

38. **tail of the row**

- *tail* : $\$^* \Rightarrow \$^*$
- prefix / strict / primitive
- let $a = \langle a_1, \ldots, a_m \rangle \in \$^*$ then:
  $tail(a) = \langle a_2, \ldots, a_m \rangle$

Row functions

39. **projection on one index**

- π_n : $\$_1 \times \ldots \times \$_n \times \$ \Rightarrow \$_n$,
  for each $n \in \mathbb{N} \backslash \{0\}$ we have such a function
- prefix / non-strict / primitive
- for $x = \langle a_1, a_2, \ldots, a_n, a_{n+1}, \ldots \rangle$ we have: $\pi_n(x) = a_n$

40. **projection on a set of indices**

- $\Pi_{(i_1,\ldots,i_k)}$: $\$_1 \times \ldots \times \$_n \times \$ \Rightarrow \$_n$, for each row $(i_1, \ldots, i_k)$ we have such a function, provided that the row is ascending and $i_k = n$
- prefix / non-strict / primitive
- for $x = \langle a_1, a_2, \ldots, a_n, a_{n+1}, \ldots \rangle$ we have:
  $\Pi_{(i_1,\ldots,i_k)}(x) = \langle a_{i_1} \ldots, a_{i_k} \rangle$

41. **Cartesian product**

- $prod : I\!F(\$_1) \times I\!F(\$_2) \Rightarrow I\!F(\$_1 \times \$_2)$
- infix / strict / derived
- $prod(a, b) :=$
  if $a = \{\}$ *then* $\{\}$
  else $iprod(pick(a), b) \cup prod(rest(a), b))$ *fi*
- auxiliary function:
  $iprod(x, b) :=$
  if $b = \{\}$ *then* $\{\}$
  else $ins((x, pick(b)), iprod(x, rest(b)))$
  fi : $\$_1 \times I\!F(\$_2) \Rightarrow I\!F(\$_1 \times \$_2)$

Tuple functions

42. **projection on one attribute**

- $\pi_\ell : [\ell : \$_1] \bowtie \$_2 \Rightarrow \$_1$,
  for each attribute $\ell \in L$ there is such a function
- prefix / non-strict / primitive
- for $x = \{\ell \mapsto a, \ldots\}$ we have: $\pi_\ell(x) = a$

43. **projection on a set of attributes**

- $\Pi_{(l_1,\ldots,l_k)} : [l_1 : \$_1, \ldots, l_k : \$_k] \bowtie \$ \Rightarrow [l_1 : \$_1, \ldots, l_k : \$_k]$
- prefix / non-strict / primitive
- for $x = \{\ell_1 \mapsto a_1, \ldots\}$ we have:
  $\Pi_{(l_{i_1},\ldots,l_{i_k})}(x) = \{\ell_{i_1} \mapsto a_{i_1}, \ldots, \ell_{i_k} \mapsto a_{i_k}\}$,
  provided that $\{\ell_{i_1}, \ldots, \ell_{i_k}\} \subset \{\ell_1, \ldots\}$

44. **tuple update**

- $\oplus : \$_1 \times \$_2 \Rightarrow \$_1 \bowtie \$_2$
- infix / non-strict / primitive
- $\{k_1 \mapsto a_1, \ldots, k_m \mapsto a_m\} \oplus \{l_1 \mapsto b_1, \ldots, l_n \mapsto b_n\} =$

$$\{r_1 \mapsto c_1, \ldots, r_p \mapsto c_p\},$$

  where $\{k_1, \ldots, k_m\} \cup \{l_1, \ldots, l_n\} = \{r_1, \ldots, r_p\}$ and
  $\forall i, j : (r_i = l_j \Rightarrow c_i = b_j) \wedge$
  $(r_i = k_j \ \wedge \ \neg\exists t : r_i = l_t) \Rightarrow c_i = a_j$

45. **join**

- $join : I\!F(\$_1 \times \$_2) \Rightarrow I\!F(\$_1 \ \bowtie \ \$_2)$
- prefix / strict / derived
- $join(x, y) :=$
  if $x = \{\}$ *then* $\{\}$
  else $ins(semijoin(pick(x), y), join(rest(x), y))$ *fi*

- auxiliary function:
  $semijoin(x, y) :=$
  $if\ x \oplus y = y \oplus x\ then\ ins(x \oplus y, semijoin(x, rest(y)))$
  $\quad else\ semijoin(x, rest(y))\ fi :$
  $\$_1 \times \mathbb{F}(\$_2) \Rightarrow \mathbb{F}(\$_1 \bowtie \$_2)$

Functions on binary relations

46. **domain**

- $dom : \mathbb{F}(\$_1 \times \$_2) \Rightarrow \mathbb{F}(\$_1)$
- prefix / strict / derived
- $dom(f) :=$
  $if\ f = \{\}\ then\ \{\}\ else\ ins(\pi_1(pick(f)), dom(rest(f)))\ fi$

47. **range**

- $rng : \mathbb{F}(\$_1 \times \$_2) \Rightarrow \mathbb{F}(\$_2)$
- prefix / strict / derived
- $rng(f) :=$
  $if\ f = \{\}\ then\ \{\}\ else\ ins(\pi_2(pick(f)), rng(rest(f)))\ fi$

48. **maximum of a relation**

- $fmax : \mathbb{F}(\$ \times \mathbb{Q}) \Rightarrow \mathbb{Q}$
- prefix / strict / derived
- $fmax(f) :=$
  $if\ f = \{\}\ then\ 0\ else\ \pi_2(pick(f))\ max\ fmax(rest(f))$

49. **set apply**

- $setapply : : \mathbb{F}(\$_1 \times \$_2) \times \$_1 \Rightarrow \mathbb{F}(\$_2)$
- prefix / strict / derived
- $setapply(f, x) := \{y : rng(f) \mid (x, y) \in f\}$

50. **apply for mappings**

- $. : \mathbb{F}(\$_1 \times \$_2) \times \$_1 \Rightarrow \$_2$
- infix / strict / derived
- $f.x := pick(setapply(f, x))$

51. **inverse**

- $inv : \mathbb{F}(\$_1 \times \$_2) \times \$_1 \Rightarrow \mathbb{F}(\$_2 \times \$_1)$
- prefix / strict / derived
- $inv(f) := \{z : prod(rng(f), dom(f)) \mid (\pi_2(z), \pi_1(z)) \in f\}$

Bibliography

[Abrial, 1974] J.R. Abrial. Data semantics. *Data Base Management*, pages 1–59, 1974. North-Holland.

[Aerts *et al.*, 1992] A.T.M. Aerts, P.M.E. de Bra, and K.M. van Hee. Transforming functional database schemes to relational representations. In *Specification of Database Systems*. Workshops in Computing Series, Springer-Verlag, 1992.

[Agha, 1986] G.A. Agha. *ACTORS, A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Ajmone Marsan *et al.*, 1985] M. Ajmone Marsan, G. Bablo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. On Petri nets with stochastic timing. In *IEEE Proceedings of the International Workshop on Timed Petri Nets*, pages 80–87, Torino, Italy, 1985.

[Andrews and Ince, 1991] D. Andrews and D. Ince. *Practical Formal Methods with VDM*. McGraw-Hill International, 1991.

[Atkinson *et al.*, 1989] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, 1989.

[Bachman, 1969] C.W. Bachman. Data structure diagrams. *Data Base 1*, 2, 1969.

[Baeten and Weijland, 1990] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[Barendregt, 1984] H.P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Studies in Logic and Foundations of Mathematics. North-Holland, 1984.

[Berthomieu and Diaz, 1991] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[Berthomieu and Menasche, 1983] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In R.E.A. Mason, editor, *IFIP, Information Processing*, volume 83, pages 41–46. Elsevier Science Publishers, 1983.

[Boardman, 1990] J. Boardman. *Systems Engineering: An Introduction.* Prentice-Hall, 1990.

[Boehm, 1981] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[Booch, 1991] G. Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[Brauer, 1980] W. Brauer. *Net Theory and Applications: Proceedings of Advanced Course on General Net Theory, Processes and Systems*, volume 84 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Brodie *et al.*, 1984] M.L. Brodie, J. Mylopoulos, and J. Schmidt. *On Conceptual Modelling: Perspective from Artificial Intelligence Databases.* Springer-Verlag, 1984.

[Buneman and Frankel, 1979] O.P. Buneman and R.E. Frankel. FQL - a functional query language. *Proceedings 1979 ACM Sigmod. International Conference on the Management of Data*, 1979.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[Ceri and Pelagatti, 1984] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[Checkland, 1981] P. Checkland. *Systems Thinking, Systems Practice.* John Wiley and Sons Ltd., 1981.

[Chen, 1976] P.P. Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, January 1976.

[Coad and Yourdon, 1990] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, 1990.

[Codd, 1970] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[Cohen *et al.*, 1986] B. Cohen, W.T. Harwood, and M.I. Jackson. *The Specification of Complex Systems.* Addison-Wesley, 1986.

[Colom and Silva, 1991] J.M. Colom and M. Silva. Convex geometry and semiflows in P/T nets, a comparative study of algorithms for

computation of minimal p-semiflows. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 79–112. Springer-Verlag, 1991.

[Dahl *et al.*, 1970] O.J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 common base language. Technical Report S-22, Norwegian Computing Center, 1970.

[Date, 1990a] C.J. Date. *An Introduction to Database Systems: Volume I.* Addison-Wesley, 5th edition, 1990.

[Date, 1990b] C.J. Date. *An Introduction to Database Systems: Volume II.* Addison-Wesley, 1990.

[David and Alla, 1989] R. David and H. Alla. *Du Grafcet aux Résaux de Petri.* Hermes-Paris, 1989.

[David and Alla, 1990] R. David and H. Alla. Autonomous and timed continuous Petri nets. *Proceedings of 11th International Conference on Applications and Theory of Petri Nets, Paris*, 1990.

[Davis and Olson, 1985] G.B. Davis and M.H. Olson. *Management.* McGraw, 2 edition, 1985.

[Di Giovanni and Iachini, 1990] R. Di Giovanni and P.L. Iachini. HOOD and Z for the development of complex software systems. In D. Bjorner and C.A.R. Hoare, editors, *VDM'90, VDM and Z - Formal Methods of Software Development*, volume 428 of *Lecture on Computer Science Notes.* Springer-Verlag, 1990.

[Dijkstra, 1968] E.W. Dijkstra. Co-operating sequential processes. *Programming Languages (F. Genuys e.d.)*, 1968. Academic Press.

[Enderton, 1977] H.B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[Falkenberg and Lindgreen, 1989] E.D. Falkenberg and P. Lindgreen, editors. *Information System Concepts: An In-depth Analysis*, IFIP TC8 Working Conference, Namur, Belgium, 1989. Elsevier Science Publishers.

[Finkel, 1990] A. Finkel. A minimal coverability graph for Petri nets. In *Proceedings of the 11th International Conference on Applications and Theory of Petri nets*, Paris, 1990.

[Galbraith, 1973] J. Galbraith. *Designing Complex Organizations.* Addisson-Wesley, Reading Mass, 1973.

[Genrich and Lautenbach, 1979] H.J. Genrich and K. Lautenbach. The analysis of distributed systems by means of predictate/transition-nets. In G. Kahn, editor, *Semantics of Concurrent Compilation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146. Springer-Verlag, 1979.

[Genrich and Lautenbach, 1981] H.J. Genrich and K. Lautenbach. System modelling with high level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Genrich, 1987] H.J. Genrich. Predictate/transition-nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986 Part I: Petri Nets, Central Models and their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[Glaser *et al.*, 1984] H. Glaser, C. Hankin, and D. Till. *Principles of Functional Programming.* Prentice-Hall International, 1984.

[Goldberg and Robson, 1983] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gyssens *et al.*, 1990] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Principles of Database Systems*, 1990.

[Hammer and McLeod, 1981] M. Hammer and D. McLeod. Data description with SDM: a semantic database model. *ACM-Transactions on Database Systems*, 6(3), 1981.

[Hayes, 1987] I. Hayes, editor. *Specification Case Studies.* Prentice-Hall, 1987.

[Hennessy, 1988] M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, 1988.

[Hesselink, 1988] W.H. Hesselink. Deadlock and fairness in morphisms of transition systems. *Theoretical Computer Science*, 59:235–257, 1988.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hopcroft and Ullmann, 1979] J.E. Hopcroft and J.D. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Hull and King, 1987] R. Hull and R. King. Semantic database modelling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3), March 1987.

[IEEE, 1989] IEEE, editor. *Petri Nets and Performance Models*, Proceedings of the 3rd International Workshop, Melbourne 1989. IEEE Computer Society Press, 1989.

[IEEE, 1991] IEEE, editor. *Petri Nets and Performance Models*, Proceedings of the 3rd International Workshop, Melbourne 1991. IEEE Computer Society Press, 1991.

[Jackson, 1983] M. Jackson. *System Development.* Prentice-Hall International, 1983.

[Jantzen and Valk, 1980] M. Jantzen and R. Valk. Formal properties of place-transition nets. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Jensen, 1990] K. Jensen. Coloured Petri nets: a high level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, 1990.

[Jensen, 1992] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* EATC Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[Jones, 1990] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1990.

[Karp and Miller, 1969] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[Kleijnen and Groenendaal, 1992] J.P.C. Kleijnen and W. van Groenendaal. *Simulation: a Statistical Perspective.* John Wiley, 1992.

[Lautenbach, 1975] K. Lautenbach. Liveness in Petri Nets. Technical report, GMD Bonn, 1975. GMD-ISF 75-02-1.

[Lewis and Papadimitriou, 1981] H.R. Lewis and Papadimitriou. *Elements of the Theory of Computing.* Prentice-Hall, 1981.

[Lundeberg *et al.*, 1981] M. Lundeberg, G. Goldkuhl, and A. Nilsson. *Information Systems Development - A Systematic Approach.* Prentice-Hall, 1981.

[Lyytinen, 1987] K. Lyytinen. Different perspectives on information systems: Problems and solutions. *ACM Computing Surveys*, 19(1), March 1987.

[Marca and McGowan, 1988] D.A. Marca and C.L. McGowan. *SADT : Structured Analysis and Design Technique.* McGraw-Hill, 1988.

[Martinez and Silva, 1982] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalised Petri net. In C. Girault and W. Reisig, editors, *Application and Theory of Petri Nets : selected papers from the first and the second European workshop*, volume 52 of *Informatik Fachberichte*, pages 301–310, Berlin, Germany, 1982. Springer-Verlag.

[Mazurkiewicz, 1984] A. Mazurkiewicz. Traces, histories, graphs: instances of a process monoid. *Mathematical Foundations of Computer Science, Lecture Notes on Computer Science*, 176:115–133, 1984. Springer.

[Meyer, 1988] B. Meyer. *Object Oriented Software Construction.* Prentice-Hall, 1988.

[Meyer, 1990] B. Meyer. *Introduction to the Theory of Programming Languages.* Prentic-Hall, 1990.

[Milner, 1980] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mintzberg, 1979] H.. Mintzberg. *The Structuring of Organisations.* Prentice-Hall, 1979.

[Murata, 1989] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Nijssen and Halpin, 1989] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach.* Prentice-Hall, 1989.

[Ören *et al.*, 1984] T.I. Ören, B.P. Zeigler, and M.S. Elzas. *Simulation and Model-based Methodologies: An Integrated Perspective*, volume 10 of *Nato ASI-series F: Computer and Systems Science.* Springer-Verlag, 1984.

[Paredaens *et al.*, 1989] J. Paredaens, P. de Bra, M. Gijssens, and D. van Gucht. *The structure of the Relational Data Model.* EATC Monographs on Theoretical Computer Science. Springer-Verlag, 1989.

[Parent and Spaccapietra, 1985] S. Parent and S. Spaccapietra. An algebra for a general entity-relationship model. *IEEE Transactions on Software Engineering*, SE-11(7), 1985.

[Peterson, 1980] J.L. Peterson. A note on coloured Petri nets. *Information Processing Letters*, 11(1):40–43, August 1980.

[Peterson, 1981] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, Prentice-Hall, 1981.

[Petri, 1962] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[Petri, 1980] C.A. Petri. Introduction to general net theory. In W. Brauer, editor, *Net Theory and Applications : Proceedings of the Advanced Course on General Net Theory, Processes and Systems*, volume 84 of *Lecture Notes in Computer Science*, pages 1–20, Hamburg, 1979, 1980. Springer-Verlag.

[Pless and Plünnecke, 1980] E. Pless and H. Plünnecke. *A Bibliography of Net Theory*, volume 80-05 of *ISF-Report*. Gesellschaft fur Mathematik und Datenverarbeitung Bonn, 2nd edition, 1980.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Pressman, 1987] R.S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 2nd edition, 1987.

[Ramamoorthy and Ho, 1980] C.V. Ramamoorthy and G.S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980.

[Reed and Roscoe, 1988] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, June 1988.

[Reisig, 1985] W. Reisig. *Petri Nets: an Introduction*. Prentice-Hall, 1985.

[Reisig, 1987] W. Reisig. Place-transition systems. *Petri Nets: Central Models and their Properties. Advances in Petri Nets 1986 Part I. LNCS 254*, 1987.

[Revuz, 1975] D. Revuz. *Markov Chains*. North-Holland/American Elsevier, 1975.

[Rishe, 1988] N. Rishe. *Database Design Fundamentals*. Prentice-Hall, 1988.

[Ross, 1977] D.T. Ross. Structured analysis: A language for communicating ideas. *IEEE Transactions on Software Engineering*, SE-3(1), 1977.

[Ross, 1983] S.M. Ross. *Stochastic Processes*. MacMillan, 1983.

[Ross, 1990] S.M. Ross. *A Course in Simulation*. Collier MacMillan, 1990.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Schek and Scholl, 1986] H.J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11:137–147, 1986.

[Schiffers and Wedde, 1978] M. Schiffers and H. Wedde. Analyzing program solutions of coordination problems by CP-nets. *Mathematical Foundations of Computer Science - Lecture Notes on Computer Science*, 64:462–473, 1978.

[Sernadas *et al.*, 1991] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large object-oriented design of information systems. In [van Assche *et al.*, 1991], 1991.

[Shannon, 1975] R.E. Shannon. *Systems Simulation : the Art and Science.* Englewood Cliffs, Prentice-Hall, 1975.

[Shipman, 1981] D.W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems*, 6:140–173, 1981.

[Sibertin-Blanc, 1991] C. Sibertin-Blanc. Cooperative objects for the conceptual modelling of organizational information systems. In [van Assche *et al.*, 1991], 1991.

[Sifakis, 1977] J. Sifakis. Use of Petri nets for performance evaluation. In H. Beilner and E. Gelenbe, editors, *Proceedings of the Third International Symposium IFIP WG. 7.3., Measuring, modelling and evaluating computer systems*, pages 75–93, Bonn-Bad Godesberg, 1977. North-Holland.

[Sifakis, 1980] J. Sifakis. Performance evaluation of systems using nets. In W. Brauer, editor, *Net theory and applications : Proceedings of the advanced course on general net theory,processes and systems*, volume 84 of *Lecture Notes in Computer Science*, pages 307–319, Hamburg, 1979, 1980. Springer-Verlag.

[Snepscheut, 1985] J.L.A. van de Snepscheut. *Trace Theory and VLSI Design.* LNCS 200. Springer-Verlag, 1985.

[Sol and van Hee, 1991] H.G. Sol and K.M. van Hee, editors. *Dynamic Modelling of Information Systems.* North-Holland, 1991.

[Sommerville, 1989] I. Sommerville. *Software Engineering.* Addison-Wesley Publishing Company, 3rd edition, 1989.

[Spaccapietra, 1987] S. Spaccapietra. *Entity-Relationship Approach: Ten Years of Experience.* North-Holland, 1987.

[Spivey, 1987] J.M. Spivey. *Understanding Z. A Specification Language and its Formal Semantics.* Cambridge University Press, 1987.

[Spivey, 1989] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[Teorey *et al.*, 1986] T.J. Teorey, D. Yang, and J.P. Frij. A logical design methodology for relational databases using the extended entity-relationship model. *Computing Surveys*, 18(2):197–222, 1986.

[Thompson, 1991] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tsichritzis and Lochovsky, 1982] D.C. Tsichritzis and F.H. Lochovsky. *Data Models.* Prentice-Hall, 1982.

[Ullman, 1988] J.D. Ullman. *Principles of Database and Knowledge-base Systems.* Computer Science Press, 1988.

[van Assche *et al.*, 1991] F.J.M. van Assche, B. Moulin, and C. Rolland, editors. *The Object Oriented Approach in Information Systems*, IFIP TC8 Working Conference, Quebec, Canada, 1991. North-Holland.

[van Benthem, 1983] J.F.A.K. van Benthem. *The Logic of Time.* D. Reidel Publishing Company, 1983.

[van der Aalst, 1992] W.M.P. van der Aalst. *Timed Coloured Petri Nets and their Application to Logistics.* PhD thesis, Eindhoven University of Technology, 1992.

[van Hee and Verkoulen, 1991] K.M. van Hee and P.A.C. Verkoulen. Integration of a data model and high-level Petri nets. In *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, pages 410–431, Gjern, Denmark, June 1991.

[van Hee and Verkoulen, 1992] K.M. van Hee and P.A.C. Verkoulen. Data, process and behaviour modelling in an integrated specification framework. In H.G. Sol and R.L. Crosslin, editors, *Proceedings of the Second International Conference on Dynamic Modelling of Information Systems*, Washington, D.C., USA, March 1992. North-Holland.

[van Hee *et al.*, 1989a] K.M. van Hee, G.J. Houben, and J.L.G. Dietz. Modeling of discrete dynamic systems — framework and examples. *Information Systems*, 14(4):277–289, 1989.

[van Hee *et al.*, 1989b] K.M. van Hee, L.J. Somers, and M. Voorhoeve. Executable specifications for distributed information systems. In [Falkenberg and Lindgreen, 1989], pages 139–156, Namur, Belgium, 1989.

[van Hee *et al.*, 1991] K.M. van Hee, L.J. Somers, and M. Voorhoeve. A formal framework for dynamic modelling of information systems. In [Sol and van Hee, 1991], pages 227–236, 1991.

[Ward and Mellor, 1985] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Yourdon, 1985.

[Wilkstrom, 1987] A. Wilkstrom. *Functional Programming using ML.* Prentice-Hall, 1987.

[Woodcock and Loomes, 1988] J. Woodcock and M Loomes. *Software Engineering Mathematics.* Pitman, 1988.

[Wordworth, 1992] J.B. Wordworth. *Software Development with Z. A pratical approach to Formal Methods in Software Engineering.* Addison-Wesley, 1992.

[Wymore, 1967] A.W. Wymore. *A Mathematical Theory of Systems Engineering: The Elements.* John Wiley and Sons Inc. New York, 1967.

[Yourdon, 1989] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

[Zeigler, 1976] B. Zeigler. *Theory of Modeling and Simulation.* Wiley Interscience, 1976.

[Zeigler, 1982] B.P. Zeigler. *Multi facetted Modelling and Discrete Event Simulation.* Academic Press, 1982.

Index

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.