EXSPECT, the functional part

Citation for published version (APA):

Document status and date:
Published: 01/01/1988

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
EXSPECT, the functional part

by

K.M. van Hee
L.J. Somers
M. Voorhoeve

88/20

December, 1988
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 or the editor.
EXSPECT, THE FUNCTIONAL PART

K.M. van Hee, L.J. Somers and M. Voorhoeve

1. INTRODUCTION

The language EXSPECT (from EXecutable SPECification Tool) has been designed for formal specification and prototyping of information systems. It has been developed to support the DES model (see [Hee]). In this paper, we concentrate on the syntax and semantics of its functional part. It is not meant as an introduction or as a reference for users (for this purpose, see [Somers]), but as a theoretical backbone proving its soundness.

This part of EXSPECT can be considered a typed functional language. In contrast to most other functional languages, the foremost construction method for creating new types is the set instead of the list. We regard a file as a set rather than a list of records. The language further combines type hierarchy (cf. [Cardelli]) with function polymorphy (cf. [Milner]).

We believe that this polymorphy and type hierarchy resemble the intuition and reasoning of mathematics. For instance, EXSPECT knows only one empty set (the empty set), not a separate empty set for every possible type. Also functions like the set-theoretical union can be applied to any two sets; the type system tells that the union of e.g. two sets of integers will give a set of integers. There does not exist a separate union for every set type, nor a "polymorphic" union that has a type as first extra argument.

EXSPECT has a modular structure. It is possible to define data types and operations on them in a module and use them in another module. The user needs no knowledge about the implementation details. In this way, "toolboxes" can be created for specific application fields and systems created out of them. In fact, a lot of elementary mathematical functions (like set union and intersection) have been created out of a handful of basic ones.

The language syntax is kept very simple. For example, all functions (like addition, union, intersection and concatenation) employ prefix notation. This makes the language easy to understand, but texts in it hard to read. There exists a "sugared" version of the language, that uses some standard conventions of programming languages.

EXSPECT has been used (by ourselves and students) for specification and prototyping and has been found adequate. However, there are some theoretical and practical shortcomings which we hope to redress in the future.

One shortcoming is that functions are regarded as expressions with parameters; a function is not an expression (and cannot be typed) unless its parameters have been given.
Thus EXSPECT is essentially first-order. It is however possible to have mappings, i.e. functions with a finite domain. Mappings are expressions (and have a type). Similarly sets (expressions) are finite, whereas types can represent infinite sets. Also user-defined type operators or recursive types are not possible, only renaming of type expressions.

The structure of the paper is as follows. We begin with a short section introducing our syntax notation. In the third section we give a syntax and semantics for types. Roughly spoken, types correspond to sets of objects. We proceed by deriving some functions operating on types that are needed later. In section 4 we give a syntax for expressions and a semantics for a special class of expressions called constants. Each constant corresponds to a dataobject. We also define two context-dependent functions \( \tau \) and \( \hat{e} \) that map expressions onto types and constants respectively. Finally, in section 5, we define the complete syntax of EXSPECT and show how a context is determined for \( \tau \) and \( \hat{e} \). We prove for an expression \( E \) that under certain conditions the semantics of \( \hat{e}(E) \) is an element of the semantics of \( \tau(E) \). We conclude with a few examples in section 6.
2. METASYNTAX

The following sections will contain several syntax descriptions, so we shall start by defining a metasyntax. This will be based on BNF, with the following peculiarities.

1. Any part of a syntax in underlined typeface is to be taken literally.

2. Any part between '{}' braces may be repeated. So 'a := {b}' is shorthand for 'a := b|b a'.

3. Any part between '[]' square brackets may be omitted. So 'a := [b]c' is shorthand for 'a := bc|c'.

4. Any part between '<>' triangular brackets may be repeated; each repetition must be preceded by a comma ','. So '< a>' is shorthand for 'a[<a>]'.

5. The syntax for identifiers (id), digits and characters is not further elaborated.
3. TYPE SYSTEM

The type system of EXSPECT is concise, yet powerful. It is possible to create types of arbitrary complexity. Types are used for structuring data; we will give an algorithm for checking the type correctness of specifications written in EXSPECT. This algorithm can be used for error detection.

In this section we first describe a set UN (for "universe") of dataobjects. Next we introduce types; in a way, a type corresponds to a subset of UN. Then we define the relation "is subtype of" between types and show that this relation implies the inclusion relation between the corresponding sets. Finally, we introduce some relations and operators upon types derived from the subtype relation. Algorithms for computing these relations and operators are given and proved.

3.1. Dataobjects

We can derive a set of dataobjects UN(A) from a set A of atomic objects. These atomic objects stand for "indivisible" concrete or abstract matters in the real world (like truth and falsehood, elementary particles or oil platforms). The atomic objects can be combined into sets or ordered pairs, giving new, composite dataobjects. We thus arrive at the following set of axioms.

Definition 3.1.1

The set UN(A) satisfies the following rules.
1. Each \( a \in A \) is an element of UN(A).
2. If \( a, b \) are elements of UN(A), then the ordered pair \( (a, b) \) is an element of UN(A).
3. If \( \text{for any } n \geq 0 \) \( a_1, ..., a_n \) are distinct elements of UN(A), then the set \( \{a_1, ..., a_n\} \) is an element of UN(A). In particular, the empty set \( \{\} \) is an element of UN(A) for each A.
4. The set UN(A) is the smallest set satisfying the three rules above.

Atomic objects present in EXSPECT are the booleans, the rational numbers and the character strings. We set UN = UN(A), where A is the above-mentioned set of atomic objects. If we add to this A a countable set \( E \) of extensions, i.e. objects outside A, we obtain the set of extended data objects EUN = UN (A U E).

Note that by the above definition, UN and EUN are sets that have but countably many elements.

3.2. Types

The types in EXSPECT are characterized by type expressions. A type expression \( (te) \) has the following syntax.

\[
te := \text{id} | \$ \text{te} | \langle \text{te } \times \text{te} \rangle | \langle \text{te } \rightarrow \text{te} \rangle
\]

For a type expression \( TE \) we define its set of identifiers \( I(TE) \) by
An identifier can have three roles. It can represent a basic type, a derived type or a type variable. The role of each identifier in a type expression is determined by a so-called type framework.

**Definition 3.2.1**

A type framework $F$ is a pair $(B, p)$, where $B$ is a (finite) set of identifiers called basic types in $F$ and $p$ (from parent) a mapping from a (finite) set of identifiers called derived types in $F$ to type expressions. The mapping $p$ may contain no cycles; in other words, for each $A$ in $\text{dom}(p)$, $A$ must have finite order. The order $\text{ord}(A)$ of a type expression $A$ is defined inductively as follows.

- $\text{ord}(A) = 0$ if $A \in B$,
- $\text{ord}(A) = 1 + \text{ord}(p(A))$ if $A \in \text{dom}(p)$,
- $\text{ord}(A) = \infty$ if $A$ is a type variable,
- $\text{ord}(A) = \max[t : I(A) \mid \text{ord}(t)]$ otherwise.

The identifiers outside $B$ and $\text{dom}(p)$ are called type variables in $F$.

A framework $F = (B, p)$ is called standard if $\{\text{Void, Bool, Num, Str}\} \subset B$. The semantics of type expressions is given by functions called distributions. Given a framework, a distribution maps each type on a set of dataobjects.

**Definition 3.2.2**

A distribution on a framework $F = (B, p)$ is a function $D$ that maps types in $F$ onto sets of extended dataobjects in such a way that

1. If $E$ is a basic type in $F$ then $D(E)$ is a set of dataobjects,
2. If $E$ is a derived type in $F$ then $D(E)$ is a subset of $D(p(E))$,
3. If $E$ is a type variable then $D(E)$ is a set of extensions,
4. If $E$ is of the form $SE'$, then $D(E)$ is the set of finite subsets of $D(E')$,
5. If $E$ is of the form $(E_1 \times E_2)$, then $D(E)$ is the set of pairs $(a, b)$, where $a$ in $D(E_1)$ and $b$ in $D(E_2)$ (the cartesian product of $D(E_1)$ and $D(E_2)$).
6. If $E$ is of the form $(E_1 \to E_2)$, then $D(E)$ is the set of finite subsets of $D(E_1) \times D(E_2)$ such that for any two different elements $(a_1, b_1)$ and $(a_2, b_2)$ in this finite subset, $a_1$ and $a_2$ are different dataobjects (the set of mappings from $D(E_1)$ to $D(E_2)$).

Please note that by part 2 of the above definition, a derived type $A$ corresponds to a subset of the set corresponding to $p(A)$.

Note that $D(T)$ is a subset of $UN$ iff $I(T) \subset B \cup \text{dom}(p)$. Types $T$ with this property are called ground types.

A distribution on a standard framework is called standard if Void, Bool, Num and Str are respectively mapped onto the empty set, the booleans, the rational numbers and the character strings.
In the sequel we suppose a standard framework \( F = (B, p) \) and a standard distribution \( D \) given. We will denote type variables by the letters \( R, S, T, U, V, \) whereas \( A, B, C, \ldots \) denote "placeholders" for arbitrary type expressions.

### 3.3. Structure induction on types

Before embarking upon the next sections, we shall unfold the principle by which the vast majority of our proofs will be given, which is complete induction on the structure of a type expression. This principle has already been used more or less in the definitions of the preceding section. If we want to prove a certain property \( P \) for all type expressions, it suffices to prove it for basic types and type variables, and then prove the following statements.

1. If \( P \) holds for \( A \), then \( P \) holds for \( \$A \),
2. If \( P \) holds for \( A \) and \( B \), then \( P \) holds for \( A \triangleright B \) and for \( A \rightarrow B \),
3. If \( A \) is derived and \( P \) holds for \( p(A) \), then \( P \) holds for \( A \).

We shall prove this assertion by induction on the order of \( A \) first and then the number of characters in the type expression \( A \).

If \( A \) is a derived type of order \( m \), then \( p(A) \) is of order \( m - 1 \). By the (order) induction hypothesis we may suppose \( P \) proven for \( p(A) \) and thus, by property 3, for \( A \).

If \( A \) is basic or variable \( P \) holds for it immediately. So \( A \) must be of the shape \( \$B \) or \( B \triangleright C \) or \( B \rightarrow C \). Our (size) induction hypothesis states that \( P \) must hold for \( B \) (in the first case) or for \( B \) and \( C \) (in the second and third). Properties 1 or 2 then establish \( P \) for \( A \).

In proving \( P \) for a type \( A \) we may thus assume that

1. If \( A \) is derived, \( P \) holds for \( p(A) \),
2. If \( A \) is composite, \( P \) holds for the types composing \( A \).

This is called the "induction hypothesis" (IH for short).

### 3.4. The subtype relation

In this section, we shall define the relation \( \triangleleft \) ("is subtype of") between types and give some properties.

#### Definition 3.4.1

The relation \( A \triangleleft B \) holds if and only if one of the following conditions holds.

(The sign \( = \) signifies syntactic equality.)

1. \( A = \text{Void} \),
2. \( A = B \),
3. \( A \) is derived and \( p(A) \triangleleft B \),
4. \( A = \$A' \) and \( B = \$B' \) and \( A' \triangleleft B' \),
5. \(A = A_1 \triangleright A_2\) and \(B = B_1 \triangleright B_2\) and \(A_1 \triangleleft B_1\) and \(A_2 \triangleleft B_2\),
6. \(A = A_1 \rightarrow A_2\) and \(B = B_1 \rightarrow B_2\) and \(A_1 \triangleleft B_1\) and \(A_2 \triangleleft B_2\),
7. \(A = A_1 \rightarrow A_2\) and \(B = B'\) and \(A_1 \triangleright A_2 \triangleleft B'\).

\(\Box\)

**Theorem 3.4.2**
If \(A \triangleleft B\), then \(D(A) \subset D(B)\).

**Proof.**
We claim that any of the conditions 1..7 guarantees that \(D(A) \subset D(B)\).

If condition 1 holds, \(D(A)\) is the empty set.

If \(A = B\), \(D(A)\) and \(D(B)\) are equal.

In case 3, we may assume (IH) that \(D(p(A)) \subset D(B)\), whereas, by Definition 3.2.2, \(D(A) \subset D(p(A))\).

In case 4, we may assume (IH) that \(D(A') \subset D(B')\), whereas \(D(A)\) and \(D(B)\) are the set of subsets of \(D(A')\) and \(D(B')\) respectively.

In case 5, we may assume (IH) that \(D(A_1) \subset D(B_1)\) and \(D(A_2) \subset D(B_2)\), proving \(D(A) \subset D(B)\) by Definition 3.2.2, sub 5.

Cases 6 and 7 are analogous, using the subcases 6 (for case 6) and 6 and 5 together (for case 7) of Definition 3.2.2. This proves our claim.

\(\Box\)

The converse of Theorem 3.4.2 is not true; we have for instance \(D(Void \triangleright A) \subset D(Void)\), since \(Void \triangleright A\) corresponds to the empty set, but not \((Void \triangleright A) \triangleleft Void\).

By extending Definition 3.4.1 we can arrive at an “iff” version of Theorem 3.4.2. This extension is of theoretical interest only.

We can also prove (by structure induction) that the subtype relation is a partial order on the set of types.

From Definition 3.4.1, an algorithm can be derived for checking whether \(A \triangleleft B\) holds for given types \(A\) and \(B\). This algorithm looks at the structure of \(A\). We now formulate a theorem, which is really no more than a reordering of Definition 3.4.1. From this theorem, a second algorithm could be derived looking at the structure of \(B\).

**Theorem 3.4.3**
Suppose \(A \neq Void\) and \(A \triangleleft B\). Then
1. \(B \neq Void\),
2. If \(B\) is basic or derived then either \(A = B\) or \(A\) is derived and \(p(A) \triangleleft B\),
3. If \(B\) is variable then \(A = B\),
4. If \(B\) is of the form \(B'\) then either \(A\) is of the form \(A'\) and \(A' \triangleleft B'\)
   or \(A\) is of the form \(A_1 \rightarrow A_2\) and \(A_1 \triangleright A_2 \triangleleft B'\),
5. If \(B\) is of the form \(B_1 \triangleright B_2\) then \(A\) is of the form \(A_1 \triangleright A_2\)
   and \(A_1 \triangleleft B_1\) and \(A_2 \triangleleft B_2\),
6. If $B$ is of the form $B_1 \rightarrow B_2$ then $A$ is of the form $A_1 \rightarrow A_2$
and $A_1 \ll B_1$ and $A_2 \ll B_2$.

Proof.
By looking carefully at Definition 3.4.1, all cases of the theorem are clear.

3.5. The least common supertype

In this subsection we shall investigate the question whether two types $A$ and $B$ have a common supertype (i.e. a type $C$ such that $A \ll C$ and $B \ll C$) and determine the least common supertype of $A$ and $B$, i.e. the type $C$ that is a subtype of all other supertypes.

Definition 3.5.1

Let $A$ and $B$ be types. Then $A$ and $B$ have a common supertype (denoted by $A \circ B$) if and only if there exists a type $C$ such that $A \ll C$ and $B \ll C$.

The following algorithm computes whether or not $A \circ B$ for any types $A$ and $B$ and computes a common supertype $\text{lcs}(A, B)$ if $A \circ B$. By convention we shall write $\text{lcs}(A, B) = \text{lcs}(C, D)$ if both exist and are equal or if not $A \circ B$ and not $C \circ D$.

Algorithm 3.5.2

1. $A \circ \text{Void}$ and $\text{Void} \circ A$ and $A \circ A$.
   
   $\text{lcs}(A, \text{Void}) = \text{lcs}(\text{Void}, A) = \text{lcs}(A, A) = A$.

   Supposing $A \neq B$, $A \neq \text{Void}$ and $B \neq \text{Void}$, we have

2. If $A$ derived and $B$ not derived or $A$ and $B$ derived and $\text{ord}(A) \geq \text{ord}(B)$,
   
   $A \circ B$ iff $p(A) \circ B$,
   
   $\text{lcs}(A, B) = \text{lcs}(p(A), B)$.

3. If $B$ derived and $A$ not derived or $A$ and $B$ derived and $\text{ord}(A) \leq \text{ord}(B)$,
   
   $A \circ B$ iff $A \circ p(B)$,
   
   $\text{lcs}(A, B) = \text{lcs}(A, p(B))$.

   Supposing also that $B$ is not derived, we have

4. If $A$ is basic or variable then not $A \circ B$.

5. $A' \circ B$ if $B = B'$ and $A' \circ B'$ or $B = B_1 \rightarrow B_2$ and $A' \circ B_1 \ll B_2$,
   
   $\text{lcs}(A', B') = \text{lcs}(A', B')$,
   
   $\text{lcs}(A', B_1 \rightarrow B_2) = \text{lcs}(A', B_1 \ll B_2)$.

6. $(A_1 \ll A_2) \circ B$ if $B = B_1 \ll B_2$ and $A_1 \circ B_1$ and $A_2 \circ B_2$,
   
   $\text{lcs}(A_1, B_1) \ll \text{lcs}(A_2, B_2)$.

7. $(A_1 \rightarrow A_2) \circ B$ if $B = B_1 \rightarrow B_2$ and $A_1 \circ B_1$ and $A_2 \circ B_2$
   
   or $B = B'$ and $(A_1 \ll A_2) \circ B'$,
   
   $\text{lcs}(A_1, B_1) \rightarrow \text{lcs}(A_2, B_2)$,
   
   $\text{lcs}(A_1 \rightarrow A_2, B') = \text{lcs}(A_1 \ll A_2, B')$.
Theorem 3.5.3

The algorithm above is correct; the function lcs computed by the algorithm has the property that if there exists a type $C$ with $A \triangleleft C$ and $B \triangleleft C$, that then $\text{lcs}(A, B) \triangleleft C$.

Proof.

It is easy to show by structure induction that if $A \triangleright B$ according to the algorithm, that then $\text{lcs}(A, B)$ is defined by the algorithm and that $A \triangleleft \text{lcs}(A, B)$ and $B \triangleleft \text{lcs}(A, B)$. So now suppose that there is a $C$ such that $A \triangleleft C$ and $B \triangleleft C$. We must show that then $A \triangleright B$ according to the algorithm and that $\text{lcs}(A, B) \triangleleft C$.

If $A = \text{Void}$ or $B = \text{Void}$ or $A = B$, this is trivial, proving (1). So we suppose from now on that $A$ and $B$ unequal and nonempty.

If $A$ and/or $B$ are derived, we are either in case (2) or (3). Suppose we are in case (2). Since $A \triangleleft C$, either $A = C$ or $p(A) \triangleleft C$ by Definition 3.4.1. If $A = C$, then $B \triangleleft A$, so by Theorem 3.4.3, $B$ is derived and $p(B) \triangleleft A$. Now either $p(B) = A$ or (again by Theorem 3.4.3) $p(B)$ is derived and $p(p(B)) \triangleleft A$. By induction we may conclude that there exists a $k > 0$ such that applying $p$ $k$ times onto $B$ we arrive at $A$. But this contradicts the assumption that the order of $B$ is not greater than the order of $A$. So we must have that $p(A) \triangleleft C$. Thus, any common supertype of $A$ and $B$ is also a common supertype of $p(A)$ and $B$. By our structure IH, $L = \text{lcs}(p(A), B)$ (as given by the algorithm) satisfies $L \triangleleft C$. But $\text{lcs}(A, B) = L$, proving (2). Case (3) is proved by interchanging $A$ and $B$. We suppose from now on that $B$ is not derived.

If $A$ is basic and/or variable and $A \triangleleft C$, then by Definition 3.4.1, $A = C$, and by Theorem 3.4.3 $B = C$. So $A = B$, contradicting our earlier assumption, proving (4).

If $A$ is of the form $\$: $A'$ and $A \triangleleft C$, then by Definition 3.4.1, $C = $ $C'$ and $A' \triangleleft C'$. If $B \triangleleft C$, then there are two subcases for $B$ by Theorem 3.4.3. The first is that $B = $ $B'$ and $B' \triangleleft C'$. By IH, $\text{lcs}(A', B') \triangleleft C'$, so by Definition 3.4.1, $\text{lcs}(A, B) \triangleleft C$. The second subcase for $B$ is that $B = B_1 \rightarrow B_2$ and $B_1 \rightarrow B_2 \triangleleft C'$. By IH, $\text{lcs}(A', B_1 \rightarrow B_2) \triangleleft C'$, so $\text{lcs}(A, B) \triangleleft C$.

If $A$ is of the form $A_1 \rightarrow A_2$ and $A \triangleleft C$, then by Definition 3.4.1, $C = C_1 \rightarrow C_2$ and $A_1 \triangleleft C_1$ and $A_2 \triangleleft C_2$. By Theorem 3.4.3, if $B \triangleleft C$, then $B = B_1 \rightarrow B_2$ and $B_1 \triangleleft C_1$ and $B_2 \triangleleft C_2$. So by IH, $A_1 \triangleright B_1$ and $A_2 \triangleright B_2$ and $\text{lcs}(A_1, B_1) \triangleleft C_1$ and $\text{lcs}(A_2, B_2) \triangleleft C_2$. By applying Definition 3.4.1 we prove (6).

If $A$ is of the form $A_1 \rightarrow A_2$ and $A \triangleleft C$, then by Definition 3.4.1 there are two possibilities for $C$. The first is that $C = C_1 \rightarrow C_2$ and $A_1 \triangleleft C_1$ and $A_2 \triangleleft C_2$. If $B \triangleleft C$, Theorem 3.4.3 prescribes $B = B_1 \rightarrow B_2$, and the argument above can be repeated. The second possibility is that $C = $ $C'$, so, if $B \triangleleft C$, Theorem 3.4.3 prescribes $B = $ $B'$ and we can interchange $A$ and $B$ and apply (5).

\[ \square \]

3.6. Valuations

A valuation is a complete function from type variables to type expressions. If $f$ is a valuation and $A$ a type, we denote by $f[A]$ the type obtained by replacing each type variable $T$ in $A$ by $f(T)$. The mapping of $A$ onto $f[A]$ is monotonic w.r.t. the partial ordering induced by the subtype relation. In fact, we have the following theorem.
Theorem 3.6.1
Let $A$ and $B$ be types with $A \prec B$. Then for every valuation $f$,  
$f[A] \prec f[B]$.  

Proof.
Straightforward by induction on the structure of $A$.  

We define the relation $\prec$ upon valuations as follows. For valuations $f$ and $g$, we say that $f$ is a subvaluation of $g$ ($f \prec g$) if $f(T) \prec g(T)$ for every type variable $T$.

The finite valuation denoted by $[T_1, ..., T_n \backslash A_1, ..., A_n]$ is the valuation $f$ satisfying $f(T_i) = A_i$ for $i = 1, ..., n$ and $f(T) = \text{Void}$ for all other type variables $T$. A special case is the empty valuation ($\text{\{}\text{\}}$).

The relation "$\prec$" is a (partial) order upon the set of valuations and $\text{\{}\text{\}}$ its minimal element.

The valuations $f$ and $g$ are said to be compatible (denoted $f \approx g$) iff there exists a valuation $h$ with $f \prec h$ and $g \prec h$. If $f$ and $g$ are compatible, we denote by $f * g$ the valuation with the property that 
\[ f * g(T) = \text{lcs}(f(T), g(T)) \]
for every type variable $T$. By Theorem 3.5.3, $f * g$ is the least common supervaluation of $f$ and $g$, i.e. for any $h$ such that $f \prec h$ and $g \prec h$, we have $f * g \prec h$.

If $f$ and $g$ (with $f \approx g$) are finite, so is $f * g$ and Algorithm 3.5.2 can be used to compute it.

3.7. Majoration

The concept of majoration bears some resemblance to resolution, i.e. binding (type) variables in a given (type) expression to (type) expressions in order to obtain a match with another given (type) expression.

Definition 3.7.1
Let $A$ and $B$ be types. Then $B$ majorates $A$ (notation $A \ll B$) iff there exists a valuation $f$ such that $A \prec f[B]$.  

The following algorithm decides for any $A$ and $B$ whether $A \ll B$ and, if so, computes a valuation $\text{mmv}(A, B)$ (the minimal majorating valuation of $B$ w.r.t. $A$) such that $A \prec \text{mmv}(A, B)[B]$.

Algorithm 3.7.2
1. Void $\ll B; \text{mmv}(\text{Void}, B) = \text{\{}\text{\}}$,  
2. If $B$ is a type variable then $A \ll B; \text{mmv}(A, B) = \text{\{}B\backslash A\text{\}}$,  
3. If $B$ is basic or derived then $A \ll B$ iff $A \prec B$; if so, $\text{mmv}(A, B) = \text{\{}\text{\}}$,  

10
4. If $A$ is derived and $B$ not derived or variable, then $A \ll B$ iff $p(A) \ll B$; if so, $\text{mmv}(A, B) = \text{mmv}(p(A), B)$.

If $A$ is not Void or derived, we have moreover

5. $A \ll \$B'$ iff either $A = \$A'$ and $A' \ll B'$
or $A = A_1 \rightarrow A_2$ and $A_1 \ll A_2 \ll B'$; if so,
$\text{mmv}(\$A', \$B') = \text{mmv}(A', B')$ and
$\text{mmv}(A_1 \rightarrow A_2, \$B') = \text{mmv}(A_1 \ll A_2, B')$.

6. $A \ll B_1 \gg B_2$ if $A = A_1 \gg A_2$ and $A_1 \ll B_1$ and $A_2 \ll B_2$
and $\text{mmv}(A_1, B_1) \approx \text{mmv}(A_2, B_2)$; if so,
$\text{mmv}(A_1 \gg A_2, B_1 \gg B_2) = \text{mmv}(A_1, B_1) \ast \text{mmv}(A_2, B_2)$.

7. $A \ll B_1 \rightarrow B_2$ iff $A = A_1 \rightarrow A_2$ and $A_1 \ll B_1$ and $A_2 \ll B_2$
and $\text{mmv}(A_1, B_1) \approx \text{mmv}(A_2, B_2)$; if so,
$\text{mmv}(A_1 \rightarrow A_2, B_1 \rightarrow B_2) = \text{mmv}(A_1, B_1) \ast \text{mmv}(A_2, B_2)$. □

**Theorem 3.7.3**

The above algorithm is correct. Moreover if $A \ll B$, then for any valuation $f$ such that $A \ll f[B]$,
$\text{mmv}(A, B) \ll f$.

**Proof.**

It is straightforward by applying Definition 3.4.5 that the conditions indicated in the algorithm are sufficient and that the valuation $\text{mmv}(A, B)$ from the algorithm satisfies $A \ll \text{mmv}(A, B)[B]$.

For the converse, suppose an $f$ given such that $A \ll f[B]$. We must show the conditions in the algorithm must hold and that $\text{mmv}(A, B) \ll f$.

If $A = \text{Void}$ or $B$ is derived or basic, this is trivial, proving (1) and (3).

If $B$ is variable, $\text{mmv}(A, B)(B) = A \ll f[B] = f(B)$ and for any other variable $X$, $\text{mmv}(A, B)(X) = \text{Void} \ll f(X)$, proving (2).

If $A$ is derived and $B$ not derived nor variable, we have either $A = f[B]$ or $p(A) \ll f[B]$. The first case is a contradiction, so the second must hold. But then $p(A) \ll B$, so (IH)
$\text{mmv}(p(A), B) \ll f$, proving (4).

So we may suppose $A$ is not derived or empty. If $B = \$B'$ then $f[B] = \$f[B']$, so $A$ must be of the form $\$A'$ or $A_1 \rightarrow A_2$. In the first case, we have $A' \ll f[B']$, so $A' \ll B'$.

Applying IH, we find
$\text{mmv}(A, B) = \text{mmv}(A', B') \ll f$.

In the second case we apply a similar argument, proving case 5.

If $B = B_1 \gg B_2$, then $f[B] = f[B_1] \gg f[B_2]$, so $A$ must be of the form $A_1 \gg A_2$ and $A_1 \ll f[B_1]$ and $A_2 \ll f[B_2]$.

Thus by IH,
$A_1 \ll B_1, A_2 \ll B_2, \text{mmv}(A_1, B_1)[B_1] \ll f[B_1], \text{mmv}(A_2, B_2)[B_2] \ll f[B_2]$.

Let us call $\text{mmv}(A_1, B_1) M_1$ and $\text{mmv}(A_2, B_2) M_2$. Since $M_1 \ll f$ and $M_2 \ll f$, $M_1$ and $M_2$ must be compatible and $M_1 \ast M_2 \ll f$, proving (6).

For (7), copy the above argument with $\gg$ replaced by $\rightarrow$. □
4. EXPRESSIONS

Expressions are represented by the following syntax.

\[
\begin{align*}
\text{expr} & : = \text{atom} \mid \text{id} \{\langle < \text{expr} > \rangle \mid \langle \text{id} \rangle \mid \text{expr} \mid \text{expr}\} \\
\text{atom} & : = \text{num-atom} \mid \text{str-atom} \\
\text{num-atom} & : = 0 \mid \text{non-0-digit} \{\text{digit}\} \\
\text{str-atom} & : = \langle\{\text{non-}^*\text{-char}\}\rangle \\
\end{align*}
\]

Atoms are the strings representing atomic dataobjects, which must have a syntax distinct from that of identifiers. In the current implementation of EXSPECT there are numerical and string atoms. In later versions it may be possible to add user-defined atoms.

We shall present an informal explanation for the diverse construction methods for expressions. The essential point is that each “good” expression represents some dataobject. The determination of this dataobject is called evaluation of the expression. The execution of an EXSPECT program essentially consists of the evaluation of certain expressions.

The evaluation of atoms is immediate; atoms correspond 1-1 to dataobjects.

An expression of the form “id” is either a placeholder for an expression defined elsewhere, a variable used for defining mappings or functions or an atomic identifier from a small set (in the current implementation of EXSPECT the only atomic identifiers are empty, true, false, quote and abort).

An expression of the form id(E₁, ..., Eₙ) signals the application of a function or parametrized expression id to the parameters E₁, ..., Eₙ. Its value is determined by applying id to (some of) the values of E₁, ..., Eₙ.

An expression of the form [x : A | B] is the function with variable x ranging over the set A defined by B. It thus represents a mapping (with domain A).

4.1. Typing of expressions

We define a partial function τ mapping expressions onto types. We call the domain of τ the set of typed expressions. The function τ is context-dependent. We want to establish a semantics s for expressions a such that s(a) ∈ D(τ(a)) under certain conditions. It is thus logical that the context for τ depends upon the distribution D (i.e. how atoms are typed) and upon the way identifiers (e.g. functions) are typed.

Definition 4.1.1

A type context consists of a pair (δ, S), where δ is a function that maps atoms onto types and S a set of tuples of the form

\[(id, L_{0}, ..., L_{n})\]

where id is an identifier, \( n \geq 0 \) and \( L_{0}, ..., L_{n} \) are types.

\[\square\]
Given a type context TC, the following recursive algorithm computes whether an expression E is typed, and computes its type \( \tau(E) \). By convention, writing \( \tau(E) = A \) for some nonsensical type expression A means that E is not typed within context TC. Also by \( A_1 << A_2 << ... << A_n \) we denote \( A_1 \) if \( n = 1 \) and \( A_1 << (A_2 << ... << A_n) \) if \( n > 1 \).

Algorithm 4.1.2
1. If E is an atom, \( \tau(E) = \delta(E) \),
2. If \( E = id \), then \( \tau(E) = ||[L] \),
   provided there is one and only one tuple of the form (id, L) in S.
3. If \( E = id(E_1, ..., E_n) \), then \( \tau(E) = \text{mmv}(A, B) [L_0] \),
   provided there exists one and only one tuple of the form (id, L_0, ..., L_n) in S
   such that \( A << B \), where \( A = \tau(E_1) << ... << \tau(E_n) \) and \( B = L_1 << ... << L_n \),
4. If \( E = [id : E_1 | E_2] \), then \( \tau(E) = A \rightarrow B \), provided \( \text{mmv}(\tau(E_1), $V) [V] = $A \) (with \( V \) a type variable) and \( \tau[id, A] (E_2) = B \).
   Here \( \tau[id, A] \) is obtained from \( \tau \) by removing all tuples of the form (id, S) (if any) from TC and adding a tuple (id, A).

Point 3 of the above algorithm is the very reason to introduce type majoration. We shall give an example of its effect. Suppose we have defined the function union for set union. The type context will then contain a tuple (union, $T, $T, $T) (with T a type variable), which means that two sets of the same set type, say $T, can be united to form a set of type, again, $T.

Suppose the type num (the rationals) has two derived types int (integers) and pos (positive rationals) and we have two expressions \( e_1 \) of type $pos and \( e_2 \) of type $int. Then the type of union(e1, e2) will be correctly $num by point 3. Note also that \( id(E_1, ..., E_n) \) cannot be typed if any expression \( E_i \) (1 \( \leq i \leq n \)) cannot be typed.

Definition 4.1.3
Given a type context TC = (\( \delta, S \)), a tuple (id, A_0, ..., A_n) for \( n \geq 0 \) is called reserved if, for any non-void types B_1, ..., B_n such that
\[ B_1 << ... << B_n \ll A_1 << ... << A_n, \]
no other tuple can be found in S with the same property.

Definition 4.1.4
A type context (\( \delta, S \)) is called standard if it contains all standard types, if \( \delta(a) < \text{Num} \) for numeric atoms a, \( \delta(a) < \text{Str} \) for string atoms a and if S contains the following reserved tuples (with T and U type variables):
\[(\text{abort, Void}), (\text{true, Bool}), (\text{false, Bool}), (\text{quote, Str}), (\text{empty, Void}), (\text{sub, Num, Num, Num}), (\text{div, Num, Num, Num}), (\text{cat, Str, Str, Str}), (\text{ins, T, T, T}), (\text{prod, T} < U, T, U).\]
In the sequel we suppose a standard type context $TC = (\delta, S)$ given.

### 4.2. Representation of data objects

We shall describe here a set of expressions, called the constants that uniquely represent the standard data objects present in EXSPECT.

**Definition 4.2.1**

The set of constants $C$ and the injective function $r$ (representation) from $UN$ to $C$ is defined as follows. The inverse of $r$ is called $s$ (semantics).

1. **Numeric constants**
   1. If $a$ is a natural number, then $r(a)$ is a numeric atom. If $c_0, c_1, \ldots, c_n$ are (in that order) the digits of $r(a)$, then $a = 10^n \cdot c_0 + \ldots + 10^0 \cdot c_n$.
   2. If $a$ is a negative integer, then $r(a) = sub(0, r(-a))$.
   3. If $a$ is a rational non-integer, then $a$ is uniquely determined by $b$ and $c$ with $a = \frac{b}{c}$, $b$ an integer, $c$ natural and $\gcd(b, c) = 1$. Then $r(a) = \text{div}(r(b), r(c))$.

2. **String constants**
   1. If $a$ is the empty string, $r(a) = \text{'}\text{'}\text{'}$.
   2. If $a$ is a string not containing quotes, $r(a) = \text{'a'}$.
   3. If $a$ is a string containing quotes, then $a$ is uniquely determined by strings $b$ and $c$ with $a = b'c$ and $b$ not containing quotes. If $b$ and $c$ are empty, $r(a) = \text{quote}$, else if $b$ is empty, then $r(a) = \text{cat}(\text{quote}, r(c))$, else $r(a) = \text{cat}(\text{'b'}, r(\text{'c'}))$.

3. **Booleans**
   1. The basic identifiers $\text{true}$ and $\text{false}$ represent respectively truth and falsehood.

4. **Sets**
   1. If $a$ is the empty set, $r(a) = \text{empty}$.
   2. If $a$ is a non-empty set, $r(a) = \text{ins}(r(o), r(a'))$, where $o \in a$ and $a' = a \setminus \{o\}$.

   The same $o$ is chosen always from $a$; how this choice is made is not specified here.

5. **Pairs**
   1. If $a = (b, c)$, then $r(a) = \text{prod}(r(b), r(c))$.

6. **Errors**
   1. The basic identifier $\text{abort}$ has no semantic equivalent.

\(\square\)

**Theorem 4.2.2**

The typing of a constant $c$ is as follows.

1. If $s(c)$ is a rational number, then $\tau(c) = \text{Num}$,
2. If $s(c)$ is a string, then $\tau(c) = \text{Str}$,
3. If $s(c)$ is a boolean, then $\tau(c) = \text{Bool}$,
4. If $c = \text{empty}$ then $\tau(c) = \text{Void}$,
5. If $c = \text{ins}(a, b)$ and $\tau(c)$ exists, then $\tau(b) = B$ for some type expression $B$ and $\tau(c) = \text{lcs}(\tau(a), B)$.
6. If $c = \text{prod}(a, b)$, then $\tau(c) = \tau(a) \times \tau(b)$.

**Proof.**
We use structure induction. If $s(c)$ is a rational number, $c$ is an atom or of the form sub(0, $b$) or of the form div(a, $b$) with $a$ and $b$ representations of rational numbers. In the first case $\tau(c) = \text{Num}$ by the condition on $\delta$. In the second case we may assume that $\tau(b) = \text{Num}$ by the IH; also $\tau(0) = \text{Num}$. Since $S$ contains the reserved tuple (sub, Num, Num, Num), $\tau(\text{sub}(0, b)) = \text{mmv}(A, B) [\text{Num}] = \text{Num}$, where

$$A = \text{Num} \times \text{Num} = \tau(0) \times \tau(b) = \text{Num} \times \text{Num} = B.$$

A similar argument holds in the third case.

If $s(c)$ is a string, $c$ is an atom or the atomic id quote or of the form cat(a, $b$), with $a$ and $b$ strings. In the first two cases the theorem is trivial by the conditions on $\delta$ and $S$ respectively. In the third case we use induction and the occurrence of (cat, Str, Str, Str) in $S$. If $s(c)$ is a boolean, the theorem is trivial by the condition on $S$.

If $C$ is the empty set, $\tau(C) = \$\text{Void}$ by the condition on $S$. If $e = \text{ins}(a, b)$, then $b = \text{empty}$ or $b = \text{ins}(d, c)$. If $\tau(c) \exists$, then $\tau(b)$ must exist, so by IH, $\tau(b) = \$B$ for some $B$. Since the tuple (ins, $\text{Str}$, $\text{T}$, $\text{T}$) is reserved in $S$, with $T$ a type variable, $c$ can be typed iff $\tau(a)$ and $\tau(b)$ exist and if

$$\tau(a) \times \tau(b) = \tau(a) \times \$B \xRightarrow{T} \$T.$$

By Algorithm 3.7.2, then $|T \setminus \tau(a)| \approx |T \setminus B|$. If so,

$$\tau(c) = |T \setminus \tau(a)| \times |T \setminus B| [\$T] = \$\text{lcs}(\tau(a), B).$$

If $c = \text{prod}(a, b)$, since the tuple (prod, $\text{T}$, $\times$, $\text{U}$, $\text{T}$, $\text{U}$) is reserved in $S$, with $T$ and $U$ type variables, we have

$$\tau(c) = \text{mmv}(P, Q) [\text{T} \times \text{U}],$$

where $P = \tau(a) \times \tau(b)$ and $Q = T \times \text{U}$.

By Algorithm 3.7.2, $\text{mmv}(P, Q) = |T, U \setminus \tau(a), \tau(b)|$, whence the theorem follows.

$\square$

**Theorem 4.2.3**
For each constant $c$ in $\text{dom}(\tau)$,

$$s(c) \in D(\tau(c))$$

**Proof.**
We set $\Gamma(c) = D(\tau(c))$. Notice that for any constant $c$, $\tau(c)$ contains no type variables, so $\Gamma(c)$ is a subset of $\text{UN}$. If $c$ is a boolean, rational or string constant, then $\tau(c) = \text{Bool}, \text{Num}$ or $\text{Str}$ respectively, so $\Gamma(c)$ is the set of booleans, rationals or strings, whereas $s(c)$ is a certain boolean, rational or string.

If $c = \text{empty}$, then $\Gamma(c)$ is the set consisting of the empty set only and $s(c)$ the empty set. If $c = \text{ins}(a, b)$ and $\tau(c) \exists$, then by Theorem 4.2.2, $\tau(c) = \$L$, with $L = \text{lcs}(\tau(a), B)$, and $\tau(b) = \$B$. $\Gamma(b)$ is the set of finite subsets of $D(B)$. If $s(b) = \{b_1, \ldots, b_n\}$ (with $n \geq 0$) then by IH, $s(b) \in \Gamma(b)$ and so $b_i \in D(B) \subset D(L)$ for $i = 1, \ldots, n$. Also $s(a) \in \Gamma(a) \subset D(L)$. So $s(c) = \{s(a), b_1, \ldots, b_n\}$ is a finite subset of $D(L)$, so $s(c) \in D(\$L) = \Gamma(c)$.

If $c = \text{prod}(a, b)$ and $\tau(c) \exists$, then by Theorem 4.2.2, $\Gamma(c)$ is the cartesian product
of $\Gamma(a)$ and $\Gamma(b)$. By IH, $s(a) \in \Gamma(a)$ and $s(b) \in \Gamma(b)$, so $s(c) = \langle a, b \rangle \in \Gamma(c)$.

\[ \square \]

### 4.3. Evaluation

In this section we present the function $\hat{e}$ that evaluates expressions. This function maps expressions onto constants. This function is complete, although there exist expressions that cannot be evaluated properly. These expressions are mapped by $\hat{e}$ onto abort. In implementations of $\hat{e}$, valuations yielding abort may not terminate. The function $\hat{e}$ is context-dependent. The context contains the semantics of functions. In cases where the context is important we write $\hat{e}_c$ if $\hat{e}$ is taken with respect to the context $c$.

**Definition 4.3.1**

A context $V$ is a function which, given an identifier $id$, types $L_1, \ldots, L_n$ and constants $c_1, \ldots, c_n$ ($n \geq 0$), produces a constant $V(id,L_1,\ldots,L_n,c_1,\ldots,c_n)$.

\[ \square \]

The following algorithm computes, for a given context $V$, the result $\hat{e}(E)$ of the evaluation of expression $E$.

**Algorithm 4.3.2**

1. If $E$ is a constant, $\hat{e}(E) = E,
2. If $E = id$, then $\hat{e}(E) = V(id),
3. If a unique tuple $(id,L_0,L_1,\ldots,L_n)$, exists in $S$ such that 
   \[ r(E_1) \prec \ldots \prec r(E_n) \prec L_1 \prec \ldots \prec L_n, \]
   then $\hat{e}(id(E_1,\ldots,E_n)) = V(id,L_1,\ldots,L_n,\hat{e}(E_1),\ldots,\hat{e}(E_n))$, 
   else $\hat{e}(id(E_1,\ldots,E_n)) = abort.$
4. If there exist $a_1,\ldots,a_n,b_1,\ldots,b_n$ in UN such that $\hat{e}(A) = r(\{a_1,\ldots,a_n\})$ with $n \geq 0$ and for $i = 1,\ldots,n$, $r(b_i) = \hat{e}[id,r(a_i)](B)$, 
   where $\hat{e}[id,a]$ is obtained from $\hat{e}$ by setting $V(id) = a,
   then $\hat{e}([id : A | B]) = r(\{a_1,b_1,\ldots,a_n,b_n\})$, else $\hat{e}([id : A | B]) = abort.$

\[ \square \]

In order to justify the type system for EXSPECT, we want to establish that expressions $E$ in $\text{dom}(\tau)$ such that $\hat{e}(E) \neq \text{abort}$ satisfy 

\[ s(\hat{e}(E)) \in D(\tau(E)). \]

We abbreviate the above property for an expression $E$ by $\Theta(E)$. Clearly, $\Theta(E)$ can only hold if the type context matches the context.

**Definition 4.3.3**

A context $V$ is called **compatible** to $TC$ iff $\Theta(E)$ holds for every expression $E$ of the form $id$ or $id(E_1,\ldots,E_n)$. 

\[ \square \]
Theorem 4.3.4

Suppose a context \( V \) is given compatible with TC. Then \( \Theta(E) \) holds for every expression \( E \).

Proof.
We use structure induction and suppose \( E \) in \( \text{dom}(\tau) \) given with \( \hat{e}(E) \neq \text{abort} \). If \( E \) is a constant, we use Theorem 4.2.3.

If \( E \) is of the form \( \text{id} \) or \( \text{id}(E_1, ... , E_n) \), we use Definition 4.3.3.

So let \( E = \text{id}(A|B) \) with \( \hat{e}(E) \neq \text{abort} \). Then \( \hat{e}(E) = \tau(\langle a_1, b_1, ... , a_n, b_n \rangle) \) for some \( a_1, ... , b_n \) in UN with \( n \geq 0 \). Choose \( i \) in \( 1, ... , n \). Then \( \tau(b_i) = \hat{e}[\text{id}, r(a_i)](B) \). On the other hand, \( \tau(E) = \alpha \rightarrow \beta \), with \( \tau(A) \triangleq \$ \alpha \) and \( \tau[\text{id}, \alpha](B) = \beta \).

By IH, \( \Theta(A) \), so \( a_i \in s(\hat{e}(A)) \in D(\tau(A)) \subseteq D(\$ \alpha) \). By Definition 3.2.2, \( D(\$ \alpha) \) is the set of finite subsets of \( D(\alpha) \), so \( a_i \in D(\alpha) \). Hence the contexts implied by \( \hat{e}[\text{id}, \tau(a_i)] \) and \( \tau[\text{id}, \alpha] \) are compatible, so (IH) \( b_i \in D(\beta) \). Hence \( \langle a_i, b_i \rangle \in D(\alpha \rightarrow \beta) \) and, since the choice of \( i \) was arbitrary, \( s(\hat{e}(E)) \in D(\$ \alpha \rightarrow \beta) \). Since \( a_1, ... , a_n \) are all different, we may conclude by Definition 3.2.2 that \( s(\hat{e}(E)) \in D(\alpha \rightarrow \beta) \).

\( \square \)

We end this section by formulating a condition that ensures the compatibility between a context and a type context.

Theorem 4.3.5

If for all tuples \( \langle \text{id}, L_0 \rangle \) in \( S \), \( s(V(\text{id})) \in D(\| L_0 \|) \) and for all tuples \( \langle \text{id}, L_0, L_1, ... , L_n \rangle \) in \( S \) and all \( A, B, M_0, M_1, ... , M_n \) with
\[
A = M_1 \gg ... \gg M_n \ll L_1 \gg ... \gg L_n = B, \quad M_0 = \text{mmv}(A, B)[L_0],
\]
and for all constants \( c_1, ... , c_n \) with \( s(c_i) \in D(M_i) \)
\[
s(V(\text{id}, L_1, ... , L_n, c_1, ... , c_n)) \in D(M_0)
\]
then \( V \) is compatible to \( TC = (\delta, S) \).

Proof.
We want to establish \( \Theta(E) \), with \( E = \text{id}(E_1, ... , E_n) \). We may assume that \( \tau(E) \) exists, so there is a (unique) \( \langle \text{id}, L_0, ... , L_n \rangle \) in \( S \) such that
\[
A = \tau(E_1) \gg ... \gg \tau(E_n) \ll L_1 \gg ... \gg L_n = B.
\]
Set \( M_0 = \text{mmv}(A, B)[L_0] \). Then \( \tau(E) = M_0 \). Set \( M_i = \tau(E_i), c_i = \hat{e}(E_i) \) \( (i = 1 ... n) \). Now \( \hat{e}(E) = V(\text{id}, L_1, ... , L_n, c_1, ... , c_n) \). By IH, \( s(c_i) \in D(M_i) \), so the condition in the theorem holds and we conclude that \( s(\hat{e}(E)) \in D(\tau(E)) \). The no-parameter case is trivial.

\( \square \)
5. LANGUAGE SYNTAX AND SEMANTICS

As stated in the introduction, EXSPECT programs can be split up into modules. The evaluation context of expressions in a program is produced by the modules together. On the other hand, each module has its own type context, so modules can be checked independently on type correctness. To ensure compatibility between type and evaluation context, modules can import the type context of other modules. We shall give the context-free syntax first, then develop some basic notions and proceed with the context-sensitive part of the syntax and semantics.

5.1. Context-free part

We present the context-free syntax for an EXSPECT module, based upon the earlier syntax definitions of an expression (expr) and type expression (te).

- **module** ::= module id ; {line ;}
- **line** ::= typedef | typedef | fundecl | fundef
- **typedef** ::= type id
g- **fundecl** ::= id [[ parlist ]] : te in id
- **parlist** ::= var : te [, parlist]
- **fundef** ::= id [[ parlist ]] defsym expr : te
- **defsym** ::= := | :=

5.2. Basic notions

In this section we develop some notions based upon the above context-free syntax. Suppose we have a function \( f \) mapping sets onto sets. We say that \( f \) is additive if \( f \) satisfies

\[
\begin{align*}
\text{additive} & \quad \text{iff } f(\{\}) = \{\}, \\
\text{additive} & \quad \text{iff } f(A \cup B) = f(A) \cup f(B).
\end{align*}
\]

An additive function is completely determined by its behaviour in singletons; from the additivity axioms we can derive its behaviour in arbitrary sets. Additive functions map sets onto sets, if there is no confusion possible we write \( f(a) = b \) instead of \( f(\{a\}) = \{b\} \) for an additive \( f \). When defining additive functions we write \( f(a) = b \) iff \( c \) meaning that \( f(\{a\}) = \{b\} \) if \( c \) holds and \( f(\{a\}) = \{\} \) otherwise. A condition \( A \) is of the form \( B \) is said to hold if the syntax variables \( (t, L, c, E, \ldots \text{ in } B) \) can be matched to actual strings present in \( A \) so that equality results.

We define the additive function \( MN \) on sets of modules such that

\[
\text{MN}(m) = \langle p, SL(L) \rangle
\]

iff \( m \) is of the form \( \text{module } p; L \), where

\[
\begin{align*}
\text{SL}(X) & = \{x\} \cup \text{SL}(Y) \\
\text{SL}(X) & = \{\}.
\end{align*}
\]

We define the additive functions \( BT, DT, CF, DF \) and \( ED \) on sets of lines \( L \) (respectively associated to the basic types, defined types, declared functions, defined functions and
external definitions in $L$) such that

$BT(l) = t$

iff $l$ is a “typedef” of the form type $t$,

$DT(l) = \{t, L\}$

iff $l$ is a “typedef” of the form type $t$ from $L$,

$CF(l) = \langle e, (L), m \rangle$

iff $l$ is an “fundecl” of the form $e : L$ in $m$,

$DF(l) = \langle e, (L), (E, x_1, \ldots, x_n) \rangle$

iff $l$ is a “fundecl” of the form $f[x_1:L_1, \ldots, x_n:L_n] : L$ in $m$,

$ED(l) = DF(l)$

iff $l$ is a “fundecl” of the form $e ::= E : L$ or $f[x_1:L_1, \ldots, x_n:L_n] ::= E : L$.

We define the function $EN$ on “expression”s (associated with the set of identifiers in them) such that

$EN(e) = \{\}$

if $e$ is an “atom”,

$EN(e) = \{(a, 0)\}$

if $e$ is an “id” of the form $a$,

$EN(e) = \{(f, n)\} \cup EN(e_1) \cup \ldots \cup EN(e_n)$

if $e$ is of the form $f(e_1, \ldots, e_n)$,

$EN(e) = EN(c) \cup (EN(d) \setminus \{(a, 0)\})$ if $e$ is of the form $[a : c | d]$.

With the above definitions we can give the context rules and semantics for EXSPECT. These context rules are divided in global and local rules. The global rules say something about the consistency of a set of modules, whereas the local rules say something about the internal consistency of a single module. The type context is defined only locally.

5.3. Local context and semantics

A module $m$ is type-consistent iff $L = \pi_2(MN(m))$ satisfies each of the following conditions.

1. The type definitions in $L$ are mappings.

$\forall [x, y : DT(L) | \pi_1(x) = \pi_1(y) \Rightarrow x = y]$ We treat $DT(L)$ as a mapping and write $\text{dom}(DT(L))$ and $DT(L) \cdot t$ with the usual meaning. The same notation is used for other sets of pairs that turn out to be mappings.

2. There is no recursion in type definitions.

$\forall [x : \text{dom}(DT(L)) | \text{ord}(x) \text{ is finite}]$, where the function ord is defined in section 3.2.

3. $BT(L)$ contains the fundamental types Void, Bool, Num and Str.

$\{\text{Void, Bool, Num, Str}\} \subset BT(L)$

If a module $m$ satisfies the above three rules, its type framework is $\langle BT(L), DT(L) \rangle$. 19
The distribution corresponding to this framework is standard and satisfies the framework requirements, but is further unknown.

The type context \((\delta, S)\) of \(m\) is as follows.

\[
\delta(x) = \text{Num} \text{ if } x \text{ is a "num-atom"},
\]

\[
\delta(x) = \text{Str} \text{ if } x \text{ is a "str-atom"},
\]

\(S\) consists of all tuples \((f, K_1, ..., K_n)\)

such that there exists a tuple \(p = (f, (K_1, K_n, ..., K_n), ...)\) in \(\text{CF}(L) \cup \text{DF}(L)\).

4. All expressions in function definitions must be correctly typed and the type of this expression must have a common supertype with the type given.

\[
(p, (K_0, K_1, ..., K_n), (E, x_1, ..., x_n)) \in \text{DF}(L) \Rightarrow K_0 \tau[x_1, K_1]...[x_n, K_n] (E).
\]

One would expect equality instead of the common supertype property \(\tau\). However, this would be too strict for practical purposes. It must be possible to specify a "weaker" type than implemented: it is good software engineering practice not to offer more than asked for. It is sometimes even necessary to "strengthen" the result type of a function; examples will be given in the next chapter. In the latter case, the designer must be sure that the function result is always of the specified type.

5.4. Global context and semantics

A set \(S\) of modules is consistent iff it satisfies the following conditions.

1. \(\text{MN}(S)\) is a mapping.

\[
\forall [p, q : \text{MN}(S)] \quad \pi_1(p) = \pi_1(q) \Rightarrow p = q
\]

We write \(L(id)\) for \(\text{MN}(S) \cdot id\).

2. Each declared function is defined in the appropriate module.

\[
\forall [a : \text{dom(MN}(S), f : \text{CF}(L(a))] \quad \exists [g : \text{ED}(\text{L}(\pi_3(f)))] \quad \pi_1(g) = \pi_1(f) \text{ and } \pi_2(g) \approx \pi_2(f)],
\]

where \(\approx\) means equality modulo a 1-1 mapping of type variables in the framework for \(\pi_3(f)\) to type variables in the framework for \(a\).

The context \(C_m\) of a module \(m\) is in general given as follows. Let \(L = L(m)\). Given a tuple \((f, (K_1, ..., K_n), p)\) in \(\text{CF}(L)\),

\[
C_m(f, K_1, ..., K_n, c_1, ..., c_n) = C_p(f, K_1, ..., K_n, c_1, ..., c_n),
\]

provided there is a tuple \((f, (K_1, ..., K_n), (E, x_1, ..., x_n))\) in \(\text{ED}(\text{MN}(L))\).

Given a tuple \((f, (K_1, ..., K_n), (E, x_1, ..., x_n))\) in \(\text{DF}(L)\),

\[
C_m(f, K_1, ..., K_n, c_1, ..., c_n) = e[x_1, c_1]...[x_n, c_n](E),
\]

where \(e\) is taken with respect to the context \(C_m\) itself.

If \(f, K_1, ..., K_n\) do not satisfy the above requirements, then for all \(c_1, ..., c_n\)

\[
C_m(f, K_1, ..., K_n, c_1, ..., c_n) = \text{abort}.
\]

As the above definition shows, the context of a module is defined in terms of itself, and the contexts of other modules. By the condition (2) above, the context \(c\) of a module satisfies a recursive equation of the form

\[
x = F \cdot x,
\]

where \(F\) is defined by

\[
(F \cdot x)(f, K_1, ..., K_n, c_1, ..., c_n) = e[x_1, c_1]...[x_n, c_n](E)
\]

if a unique tuple of the form \((f, (K_1, ..., K_n), ...)\) is in \(\text{DF}(L) \cup \text{CF}(L)\) and

\[
(F \cdot x)(f, ..., c_n) = \text{abort}
\]

otherwise. The context \(c\) is the least fix point for this equation (cf. [Schmidt]).
This fix point exists, for the set of contexts forms a cpo; for contexts \( c, d \) we define
\[ c \sqsubseteq d \text{ iff for all } f, K_1, ..., K_n, c_1, ..., c_n \text{ such that} \]
\[ c(f, K_1, ..., K_n, c_1, ..., c_n) \neq \text{abort}, \]
we have
\[ c(f, K_1, ..., K_n, c_1, ..., c_n) = d(f, K_1, ..., K_n, c_1, ..., c_n). \]
The "bottom" element of this cpo is the trivial context whose value is abort everywhere. Also, an ascending sequence of contexts has an upper limit which is also a context. The function \( F \), mapping contexts to contexts, defined by
\[ F \cdot c(f, K_1, ..., K_n, c_1, ..., c_n) = e_c[x_1, c_1], ..., [x_n, c_n](E) \]
is clearly continuous.

The starting point for the above "bootstrapping" process is the (virtual) module basic. This module has a fixed context. All evaluable EXSPECT expressions can be expanded into an expression containing only these basic identifiers. So the above fix point is determined by successive approximation, starting from the basic context.

The basic id's are presented in a list. For each id we give a short description of its meaning, its parameter types (determining the basic context) and its result type (needed in the declarations in other modules). The first ten id's we already encountered; they were necessary for the construction of constants. When the short description is preceded by a star (*), the meaning is further elaborated below the list. Unless explicitly stated otherwise, a function result is abort if any argument evaluates to abort or if there is a type mismatch. Note that for the id's listed below, the context and type context are compatible.

<table>
<thead>
<tr>
<th>id</th>
<th>meaning</th>
<th>result type</th>
<th>parameter types</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>error (not in UN)</td>
<td>Void</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>truth</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>falsehood</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td>quote</td>
<td>the string '</td>
<td>Str</td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td>empty set</td>
<td>$Void</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>subtraction</td>
<td>Num</td>
<td>Num, Num</td>
</tr>
<tr>
<td>div</td>
<td>* division</td>
<td>Num</td>
<td>Num, Num</td>
</tr>
<tr>
<td>cat</td>
<td>concatenation</td>
<td>Str</td>
<td>Str, Str</td>
</tr>
<tr>
<td>ins</td>
<td>insertion in set</td>
<td>$T</td>
<td>T, $T</td>
</tr>
<tr>
<td>prod</td>
<td>pairing</td>
<td>T &gt;\times S</td>
<td>T, S</td>
</tr>
<tr>
<td>cond</td>
<td>* if-then-else construction</td>
<td>T</td>
<td>Bool, T, T</td>
</tr>
<tr>
<td>eq</td>
<td>equality test</td>
<td>Bool</td>
<td>T, T</td>
</tr>
<tr>
<td>pos</td>
<td>positivity test</td>
<td>Bool</td>
<td>Num</td>
</tr>
<tr>
<td>head</td>
<td>* first char</td>
<td>Str</td>
<td>Str</td>
</tr>
<tr>
<td>tail</td>
<td>* all but head</td>
<td>Str</td>
<td>Str</td>
</tr>
<tr>
<td>pick</td>
<td>* selection from set</td>
<td>T</td>
<td>$T</td>
</tr>
<tr>
<td>rest</td>
<td>* all but pick</td>
<td>$T</td>
<td>$T</td>
</tr>
<tr>
<td>pi1</td>
<td>projection 1st coordinate</td>
<td>T</td>
<td>T &gt;\times S</td>
</tr>
<tr>
<td>pi2</td>
<td>projection 2nd coordinate</td>
<td>S</td>
<td>T &gt;\times S</td>
</tr>
</tbody>
</table>

\[ \hat{e}(\text{div}(a, b)) = \text{abort if } \hat{e}(b) = 0. \]
\[ \hat{e}(\text{cond}(a, b, c)) = \hat{e}(b) \text{ if } \hat{e}(a) = \text{true}, \text{ even if } \hat{e}(c) = \text{abort}, \]
\[ \hat{e}(\text{cond}(a, b, c)) = \hat{e}(c) \text{ if } \hat{e}(a) = \text{false}, \text{ even if } \hat{e}(b) = \text{abort}. \]
\[ e(\text{head}(a)) = e(\text{tail}(a)) = \text{abort} \text{ if } e(a) = "" \text{ (the empty string)}, \]
\[ e(\text{cat}(\text{head}(a), \text{tail}(a))) = e(a) \text{ if } e(a) \text{ is any non-empty string}. \]
\[ e(\text{pick}(a)) = e(\text{rest}(a)) = \text{abort} \text{ if } e(a) = \text{empty}, \]
\[ e(\text{pick}(a)) = \alpha, e(\text{rest}(a)) = a' \text{ if } e(a) = \text{ins}(\alpha, a'). \]

Apart from "basic", other virtual modules may be included.

### 5.5. Context compatibility

In this section we describe how a consistent set of modules can be constructed, i.e. such that for each module the (evaluation) context and type context are compatible. Our starting point is the basic module which is consistent.

**Theorem 5.5.1**

Suppose \( S = \{M_1, \ldots, M_n\} \) is a consistent set of modules. Then \( S' = S \cup \{M\} \) is also consistent if \( M \) satisfies the following conditions. (We write \( L \) for \( \text{L}(M) \) and \( L(m) \) for \( \text{L}(S) \cdot m \).)

1. The module name is new.
   \[ \pi_1(MN(M)) \notin \text{dom}(MN(S)). \]
2. All declared id's in \( L \) are defined in the corresponding module in \( S \) and \( \text{BT}(L) \) contains all types that are used in a declaration and that are basic or defined in the corresponding module.
   \[ \forall \{f, t, m\} \in \text{CF}(L) \exists \{p : \text{ED}(m) | \pi_1(p) = f \text{ and } \pi_2(p) \approx t\} \text{ and} \]
   \[ \forall \{P : (\bigcup[i : 0, \ldots, 1] \{(\pi_1(t)) \cap (\text{BT}(L(m))) \cup \text{dom}(\text{DT}(L(m))) | P \in \text{BT}(L))\}. \]

Here \( I(A) \) is the set of identifiers in the type expression \( A \) as defined in section 3.2.

3. \( M \) is type consistent.

4. All defined functions of which the typing is strengthened must satisfy the conditions of Theorem 4.3.5.

   \[ \forall \{\xi : \text{DF}(L) | \xi = \langle f, \langle K_0, K_1, \ldots, K_n \rangle, \langle E, x_1, \ldots, x_n \rangle \} \Rightarrow (\tau[x_1, K_1 | \ldots | x_n, K_n](E) \triangleq K_0 \text{ or } P(\xi)). \]

Here the condition \( P \) means that for all type expressions \( A, B, H_0, H_1, \ldots, H_n \) and constants \( c_1, \ldots, c_n \) with

\[ A = H_1 \gg \ldots \gg H_n \ll K_1 \gg \ldots \gg K_n = B \text{ and} \]
\[ H_0 = \text{mmv}(A, B)[K_0] \text{ and } s(c_i) \in \text{DF}(H_i), (i = 1, \ldots, n) \]

we must have that

\[ s(e[x_1, c_1 | \ldots | x_n, c_n](E)) \in \text{DF}(H_0). \]

**Proof.** Conditions 1 and 2 imply respectively conditions 1 and 2 of section 5.4. The second part of condition 2 ensures that types that are variable in \( M \) and that occur in \( \text{DF}(L) \) are also variable in the corresponding module so that the equivalence implied by \( \approx \) can be realized. The local type consistency is guaranteed by condition 3. So we are left with the compatibility of context and type context.

As we pointed out in the previous section, the context implied by \( L \) is the fix point of a function \( F \) mapping contexts to contexts. This function is defined by

\[ F \cdot c(f, K_1, \ldots, K_n, c_1, \ldots, c_n) = e_c[x_1, c_1 | \ldots | x_n, c_n](E) \]

for some expression \( E \), if a unique tuple of the form \( \langle f, \langle K_0, K_1, \ldots, K_n \rangle, \ldots \rangle \) can be found in \( \text{DF}(L) \cup \text{CF}(L) \) and otherwise.

---

22
\[ F \cdot c(f, \ldots, c_n) = \text{abort.} \]

We assert that if \( c \) is compatible to the type context \( X \) implied by \( L \), the same holds for \( F \cdot c \). Let
\[
\Phi = \text{Fix}(F) = \bigsqcup \{ i : 1, 2, \ldots \mid F^i \cdot \bot \}
\]
where \( \bot \) is the trivial context. Now \( \bot \) is trivially compatible to \( X \), so by induction \( F^i \cdot \bot \) for each \( i \) is compatible to \( X \). Let \( E \) be an expression. Then \( s(\hat{c}_\Phi(E)) = s(\hat{c}_P(E)) \), where \( P = F^i \cdot \bot \) for some \( i \). \( P \) is compatible, so \( s(\hat{c}_\Phi(E)) \in D(\tau(E)) \). Since \( E \) was arbitrarily chosen, \( \Phi \) is compatible.

Now we must prove our assertion. If the tuple stems from \( CF(L) \), the context equation stems from another module, which we supposed to be consistent. So the tuple stems from \( DF(L) \). If there is a type strengthening, condition 4 applies. Now let \( (f, \langle K_0, \ldots, K_n \rangle, \langle E, x_1, \ldots, x_n \rangle) \in DF(L) \) and \( \tau[x_1, K_1] \ldots [x_n, K_n](E) \not< K_0 \).

We want to establish the conditions in Theorem 4.3.5 here, so let \( A, B, H_1, \ldots, H_n, c_1, \ldots, c_n \) be given such that
\[
A = H_1 > < \ldots > < H_n > < K_1 > < \ldots > < K_n = B, \quad H_0 = \text{mmv}(A, B)[K_0] \text{ and } s(c_i) \in D(H_i) \text{ for } i = 1, \ldots, n.
\]
By the definition of majoration and Algorithm 3.7.2, the valuation \( f = \text{mmv}(A, B) \) satisfies
\[
H_i < f[K_i] \text{ for } i = 1, \ldots, n, \quad H_0 = f[K_0].
\]
By Theorem 3.6.1, \( \tau[x_1, f[K_1]] \ldots [x_n, f[K_n]](E) \not< f[K_0] = H_0 \), so
\[
\tau[x_1, H_1] \ldots [x_n, H_n](E) \not< H_0.
\]
We conclude that the context and type context implied by \( \hat{c}_e[x_1, c_1] \ldots [x_n, c_n] \) and \( \tau[x_1, H_1] \ldots [x_n, H_n] \) are compatible, so
\[
F \cdot (f, K_0, \ldots, K_n, c_1, \ldots, c_n) \in D(\tau[x_1, H_1] \ldots [x_n, H_n](E)) \subset D(H_0).
\]
The conditions in Theorem 4.3.5 hold in this case, so our assertion is true.

\( \Box \)
6. EXAMPLE

The following module contains a few example definitions. Everything between the -- and the linebreak is skipped as comment.

module ex;

type Num;
type Str;
type Bool;
type Void;

empty:$Void in basic;
cond[x:Bool,y:T,z:T]: T in basic;
eq[x:T,y:T]: Bool in basic;
ins[x:T,y:$T]: $T in basic;
pick[x:$T]: T in basic;
rest[x:$T]: $T in basic;
π₁[x:T><S]: T in basic;
π₂[x:T><S]: S in basic;

-- set union

union[x:$T,y:$T] := cond (eq(x,empty), y,
  ins (pick(x), union(rest(x),y))): $T;

-- function application

apply[m:T->S,x:T] := cond (eq(π₁(pick(m)),x), π₂(pick(m)),
  apply(mrest(m),x)): S;
mrest[m:T->S] := rest(m): T->S; ...... type strengthening

-- application to a set

apply[m:T->S,x:$T] := cond (eq(x,empty), empty,
  ins (apply(m,pick(x)), apply(m,rest(x)))): $S;

The set union example shows a recursive definition. It is easy to show by induction,
that if s(\tilde{e}(x)) and s(\tilde{e}(y)) exist and are sets and if a ∈ UN,

(a ∈ s(\tilde{e}(x)) or a ∈ s(\tilde{e}(y))) iff a ∈ s(\tilde{e}(\text{union}(x,y))).

We shall omit the s and \tilde{e} functions and write a = b if \tilde{e}(a) = \tilde{e}(b) or s(\tilde{e}(a)) = b or
a = s(\tilde{e}(b)). The above assertion becomes

(a ∈ x or a ∈ y) iff a ∈ \text{union}(x,y). Now suppose that x = {}. In that case
\text{union}(x,y) = y by the definition of \text{eq} and \text{cond}, making the above assertion true. So
let \( x \neq \{ \} \). Then 
\[
\text{union}(x,y) = \text{ins}(\text{pick}(x), \text{union}(\text{rest}(x),y)).
\]
If \( a = \text{pick}(x) \), then \( a \in x \) and \( a \in \text{ins}(\text{pick}(x),...) \), proving the assertion. If not, by the definition of \( \text{rest} \),
\[
a \in x \text{ iff } a \in x',
\]
where \( x' = \text{rest}(x) \). Now \( x' \) has less elements than \( x \), so we may apply the induction hypothesis and conclude
\[
a \in x' \text{ or } a \in y \text{ iff } a \in \text{union}(x',y).
\]
Combining these last conclusions completes our proof.

Note that the result of \( \text{union}(x,y) \) does not depend on how the function \( \text{pick} \) is implemented. This is of course the right way to use \( \text{pick} \).

The second example shows another recursive definition for application of a mapping. We can show that if \( m \) and \( x \) exist and \( m \) is a mapping,
\[
\text{apply}(m,x) = y \text{ if } (x,y) \in m, \text{ apply}(m,x) = \text{abort} \text{ if } (x,y) \notin m.
\]
So again the definition is independent of the implementation of \( \text{pick} \). The proof uses the fact that \( m \) is a mapping and not merely a set of pairs.

Note the use and definition of the auxiliary function \( \text{mrest} \), that is nothing but a redefinition of \( \text{rest} \) for mappings. If we replace \( \text{mrest} \) by \( \text{rest} \) in the definition of \( \text{apply} \), the definition is incorrectly typed. By the type context for \( \text{rest} \), \( \tau(\text{rest}(m)) = \mathcal{S} \cdot \mathcal{T} \), so \( \text{apply}(\text{rest}(m),x) \) cannot be typed, because \( \mathcal{S} \cdot \mathcal{T} \cdot \mathcal{S} \cdot \mathcal{T} \) is not majorated by \( \mathcal{S} \cdot \mathcal{T} \cdot \mathcal{T} \).

This makes sense, since the type checking algorithm cannot know that deleting a pair from a mapping produces another mapping. It is a "theorem" the programmer knows; he tells it to EXSPECT by defining the function \( \text{mrest} \) with a "stronger" result type than expected.

The third example shows that the same name can be used for different functions (overloading). The second application applies a function to a subset of its domain and returns a subset of its range. In the expression
\[
\text{ins}(\text{apply}(m,\text{pick}(x)), \text{apply}(m,\text{rest}(x)))
\]
in the definition, the typing of the parameters ensures that the first apply is the element application, whereas the second is a recursive call of the set application itself.
7. REFERENCES

   private publication


Cardelli, L.: A Semantics of Multiple Inheritance.
   Proceedings Symposium on Semantics of Data Types, pp. 51-67
   Lecture Notes in Computer Science 173, Springer Verlag, 1984

Hee, K.M. van, G.J. Houben, L.J. Somers and M. Voorhoeve:
   Executable specification for discrete event systems.
   Computer Science Notes 88/17 (1988)

In this series appeared:

<table>
<thead>
<tr>
<th>No.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The formal specification and derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a computer for evaluation of flow films</td>
</tr>
<tr>
<td>85/04</td>
<td>T. Verhoeff, H.M.J.L. Schols</td>
<td>Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems</td>
</tr>
<tr>
<td>86/02</td>
<td>G.A. Bussing, K.M. van Hee, M. Voorhoeve</td>
<td>ELISA, A language for formal specifications of information systems</td>
</tr>
<tr>
<td>86/03</td>
<td>Rob Hoogerwoord</td>
<td>Some reflections on the implementation of trace structures</td>
</tr>
<tr>
<td>86/04</td>
<td>G.J. Houben, J. Paredaens, K.M. van Hee</td>
<td>The partition of an information system in several parallel systems</td>
</tr>
<tr>
<td>86/05</td>
<td>Jan L.G. Dietz, Kees M. van Hee</td>
<td>A framework for the conceptual modeling of discrete dynamic systems</td>
</tr>
<tr>
<td>86/06</td>
<td>Tom Verhoeff</td>
<td>Nondeterminism and divergence created by concealment in CSP</td>
</tr>
<tr>
<td>86/07</td>
<td>R. Gerth, L. Shira</td>
<td>On proving communication closedness of distributed layers</td>
</tr>
<tr>
<td>86/09</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Full abstraction of a real-time denotational semantics for an OCCAM-like language</td>
</tr>
<tr>
<td>86/10</td>
<td>J. Hooman</td>
<td>A compositional proof theory for real-time distributed message passing</td>
</tr>
<tr>
<td>86/11</td>
<td>W.P. de Roever</td>
<td>Questions to Robin Milner - A responder's commentary (IPIP86)</td>
</tr>
<tr>
<td>86/12</td>
<td>A. Boucher, R. Gerth</td>
<td>A timed failures model for extended communicating processes</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>86/14</td>
<td>R. Koymans</td>
<td>Specifying passing systems requires extending temporal logic</td>
</tr>
<tr>
<td>87/01</td>
<td>R. Gerth</td>
<td>On the existence of sound and complete axiomatizations of the monitor concept</td>
</tr>
<tr>
<td>87/02</td>
<td>Simon J. Klaver, Chris F.M. Verberne</td>
<td>Federatieve Databases</td>
</tr>
<tr>
<td>87/03</td>
<td>G.J. Houben, J.Paredaens</td>
<td>A formal approach to distributed information systems</td>
</tr>
<tr>
<td>87/04</td>
<td>T. Verhoeff</td>
<td>Delay-insensitive codes - An overview</td>
</tr>
<tr>
<td>87/05</td>
<td>R. Kuiper</td>
<td>Enforcing non-determinism via linear time temporal logic specification.</td>
</tr>
<tr>
<td>87/06</td>
<td>R. Koymans</td>
<td>Temporele logica specificatie van message passing en real-time systemen (in Dutch).</td>
</tr>
<tr>
<td>87/07</td>
<td>R. Koymans</td>
<td>Specifying message passing and real-time systems with real-time temporal logic.</td>
</tr>
<tr>
<td>87/08</td>
<td>H.M.J.L. Schols</td>
<td>The maximum number of states after projection.</td>
</tr>
<tr>
<td>87/10</td>
<td>T. Verhoeff</td>
<td>Three families of maximally nondeterministic automata.</td>
</tr>
<tr>
<td>87/11</td>
<td>P. Lemmens</td>
<td>Eldorado ins and outs. Specifications of a database management toolkit according to the functional model.</td>
</tr>
<tr>
<td>87/12</td>
<td>K.M. van Hee and A. Lapinski</td>
<td>OR and AI approaches to decision support systems.</td>
</tr>
<tr>
<td>87/13</td>
<td>J.C.S.P. van der Woude</td>
<td>Playing with patterns, searching for strings.</td>
</tr>
<tr>
<td>87/14</td>
<td>J. Hoorman</td>
<td>A compositional proof system for an occam-like real-time language</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>87/15</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>A compositional semantics for statecharts</td>
</tr>
<tr>
<td>87/16</td>
<td>H.M.M. ten Eikelder, J.C.F. Wilmont</td>
<td>Normal forms for a class of formulas</td>
</tr>
<tr>
<td>87/17</td>
<td>K.M. van Hee, G.-J. Houben, J.L.G. Dietz</td>
<td>Modelling of discrete dynamic systems framework and examples</td>
</tr>
<tr>
<td>87/18</td>
<td>C.W.A.M. van Overveld</td>
<td>An integer algorithm for rendering curved surfaces</td>
</tr>
<tr>
<td>87/19</td>
<td>A.J. Seebregts</td>
<td>Optimalisering van file allocatie in gedistribueerde database systemen</td>
</tr>
<tr>
<td>87/20</td>
<td>G.J. Houben, J. Paredaens</td>
<td>The $R^2$-Algebra: An extension of an algebra for nested relations</td>
</tr>
<tr>
<td>87/21</td>
<td>R. Gerth, M. Codish, Y. Lichtenstein, E. Shapiro</td>
<td>Fully abstract denotational semantics for concurrent PROLOG</td>
</tr>
<tr>
<td>88/01</td>
<td>T. Verhoeff</td>
<td>A Parallel Program That Generates the Möbius Sequence</td>
</tr>
<tr>
<td>88/02</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specification for Information Systems</td>
</tr>
<tr>
<td>88/03</td>
<td>T. Verhoeff</td>
<td>Settling a Question about Pythagorean Triples</td>
</tr>
<tr>
<td>88/04</td>
<td>G.J. Houben, J. Paredaens, D. Tahon</td>
<td>The Nested Relational Algebra: A Tool to handle Structured Information</td>
</tr>
<tr>
<td>88/05</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>Executable Specifications for Information Systems</td>
</tr>
<tr>
<td>88/06</td>
<td>H.M.J.L. Schols</td>
<td>Notes on Delay-Insensitive Communication</td>
</tr>
<tr>
<td>88/07</td>
<td>C. Huizing, R. Gerth, W.P. de Roever</td>
<td>Modelling Statecharts behaviour in a fully abstract way</td>
</tr>
<tr>
<td>88/08</td>
<td>K.M. van Hee, G.J. Houben, L.J. Somers, M. Voorhoeve</td>
<td>A Formal model for System Specification</td>
</tr>
<tr>
<td>88/09</td>
<td>A.T.M. Aerts, K.M. van Hee</td>
<td>A Tutorial for Data Modelling</td>
</tr>
</tbody>
</table>
88/10  J.C. Ebergen          A Formal Approach to Designing Delay Insensitive Circuits
88/11  G.J. Houben
      J.Paredaens          A graphical interface formalism: specifying nested relational databases
88/12  A.E. Eiben          Abstract theory of planning
88/13  A. Bijlsma          A unified approach to sequences, bags, and trees
88/14  H.M.M. ten Eikelder
      R.H. Mak              Language theory of a lambda-calculus with recursive types
88/15  R. Bos
      C. Hemerik          An introduction to the category theoretic solution of recursive domain equations
88/16  C.Hemerik
      J.P.Katoen          Bottom-up tree acceptors
88/17  K.M. van Hee
      G.J. Houben
      L.J. Somers
      M. Voorhoeve         Executable specifications for discrete event systems
88/18  K.M. van Hee
      P.M.P. Rambags       Discrete event systems: concepts and basic results.
88/19  D.K. Hammer
      K.M. van Hee          Fasering en documentatie in software engineering.
88/20  K.M. van Hee
      L. Somers
      M. Voorhoeve          EXSPECT, the functional part.