The Static Part of
the Design Language COLD-K

Gerard R. Renardel de Lavalette
Department of Computing Science, University of Groningen,
P.O. box 800, 9700 AV Groningen, the Netherlands
E-mail: grl@cs.rug.nl

Abstract

This paper is about the static fragment of the design language
COLD-K, obtained by dropping all dynamic features (procedures
and expressions). It contains definitions of syntax and semantics,
together with a presentation of the notions used in the definition
of the semantics, such as $MPL_{\omega}$ (many-sorted partial infinitary
logic), inductive definitions, the algebra of theories (with the operations
renaming, import and export) and the type structure defined over
this algebra.

1 Introduction

COLD-K is a wide spectrum design language, developed at Philips Research Eind-
hoven (the Netherlands), mainly in the ESPRIT project METEOR. In this paper
we give a survey of several – not all – features of COLD-K with an emphasis on
their semantics. These features are:

- algebraic expressions (denoting objects) containing partial functions and
descriptions;
- assertions over objects as in first order many-sorted predicate logic with
equality;
- both explicit and inductive definitions of functions and predicates;
- the modularisation constructs import, export and renaming;
- parametrisation by lambda abstraction, involving an implementation rela-
tion.

The fragment of COLD-K consisting of these features is baptized COLD-K$^2$. The
main features of COLD-K not in COLD-K$^2$ are:

- multivalued functions;
- overloading;
- dynamic features involving state changes (statements, procedures with side
effects, modification of functions and predicates);
- the notions of origin and origin consistency.

1.1 Survey of the rest of this paper

In 2, we present some background information on COLD (some history, design
decisions, the status of this paper). 3 is devoted to the syntax and semantics of
COLD-K$^2$ and related theory. Some more technical information is presented in
the Appendix, and also a few remarks on wellformedness (sometimes called the static semantics).

1.2 Acknowledgements

The research for this paper was partly performed in close and animated collaboration with Loet Feijs, Hans Jonkers, Karst Koymans and Job Zwiers. Many valuable comments on a draft version were given by Kees Middelburg. Jan Bergstra suggested to consider the static fragment of COLD-K.

2 Background information

2.1 Some history

The main designer of COLD is Hans Jonkers. In 1983, in a joint project of Philips Research and Philips Telecommunication Industry, the idea of a single linguistic framework emerged. This was worked out by Jonkers to a prototype for a design language called COLD, for CHILL-Oriented Language for Design (CHILL is a programming language frequently used for telecommunication applications); following the CHILL-independent development of the language, the meaning of the acronym was changed into Common Object-oriented Language for Design; it is described in [13]. In the ESPRIT pilot project FAST, a simplified version called COLD-S ($S$ for sequential) was developed. FAST has been succeeded by the ESPRIT project METEOR, in which the development of COLD was pursued further, leading to an elaborated and detailed formal definition of COLD-K and its semantics in [11] which forms the basis and main reference for this paper.

Some case studies involving the successive versions of COLD have been published (e.g., [6], [7], [9], [22]). A textbook on formal specification and design based on COLD has recently appeared ([10]). A user-oriented version, COLD-1, is currently under development: see e.g. [14].

COLD-K has served as a model for other specification languages, notably VVSL (the VIP VDM Specification Language, see [17], [18]) and PSF (Process Specification Formalism, see [1]).

2.2 Design decisions in the development of COLD

The main design decisions in the development of COLD-K are:

- the choice for a wide-spectrum language, covering not only formal specifications but also design and implementation (this led to the incorporation of imperative programming constructs);
- an important role for the logical semantics during the design of the language;
- algebraic specifications as a starting point, but extending equational logic to full first-order predicate logic with equality;
- partial functions combined with two-valued logic, inspired by E-logic [24];
- an assertion language for programs based on a variant of Dynamic Logic [13];
- inductive definitions of functions and predicates;
- modularisation inspired by Module Algebra [4];
- parametrisation by lambda abstraction of free module terms, with an important role for the implementation relation between modules;
- language constructs supporting the notion of software components.
2.3 Overview of COLD

The central notion in COLD is the class concept. In COLD-K, a class can be seen as an abstract machine with a collection of states. These states are many-sorted algebras with predicates and (partial) functions, all states of a class having the same signature. States can be modified nondeterministically by procedures: the basic change actions are the creation of a new object and the modification of a function or a predicate. Moreover a procedure can return one or more objects. In COLD-K², however, no dynamic features are present and a class is just a many-sorted algebra.

Classes can be described by giving definitions of the sorts, functions and predicates of its states and of the procedures that can modify the states. These class descriptions can be subjected to modularisation and parametrisation operations, and this results in schemes. Finally schemes can be combined to form components which constitute systems.

2.4 Status of this paper

In the language COLD-K² presented in this paper, we have combined language features of COLD-K which have a rather well-understood semantics (see the first part of this Introduction). Most of the material presented here is based on work reported in other publications which we mention here.

- [11], the reference report of COLD-K, containing the full definition of syntax and semantics. The language definition is preceded by chapters on some of the fundamentals behind the semantical domains: the logic MPLα (for the semantics of assertions, expressions and inductive definitions), the Class Algebra (for the semantics of modularisation constructs) and a version of lambda calculus called λπ (for the semantics of parametrisation).
- [16] about the logic MPLα, an adapted version of the corresponding chapter of [11].
- [21] on the algebra of theories, a related alternative for Class Algebra as the semantical domain for the modularisation constructs.
- [23] on the semantics of the implicit inductive definitions used in COLD.

This is a condensed version of [20]. The theory behind some dynamical aspects of COLD-K is studied in [12].

The following parts of this paper contain new material:

- 3.3.5 on the semantics of implicit inductive definitions of functions (only sketched in [23]);
- 3.5.3 on the type structure over the algebra of theories.
3 COLD-K²

3.1 The description of syntax and semantics

The overall structure of COLD-K² can be rendered as follows: the terms refer to sublanguages.

```
DESIGN
   |
PARAMETERIZED SCHEME
   |
SCHEME
   |
DEFINITION
/ \ 
ASSERTION EXPRESSION
```

In the subsequent subsections, we present these sublanguages. After the definition of the syntax in the form of a BNF-grammar, the semantic domain is discussed and finally the semantics is defined.

The definition of COLD-K² given here is presented in the form of a BNF-grammar, where | separates alternatives, [X] denotes zero or one occurrence of X, and {X} denotes one or more occurrences of X. We add the following meta-rules:

- `<xxxx-list> :: = [<xxxx>]`
- `<xxxx-list '*' > :: = [<xxxx>] {*<xxxx>}]
  where * is some symbol acting as delimiter
- `<'*'<xxxx-list '*' pod '*''> :: = [*<xxxx> {**<xxxx>}**]`
- `<xxxx-var> :: = <identifier>`
- `<xxxx-name> :: = <identifier>`

3.2 The expression language and the assertion language

3.2.1 Introduction.

Expressions describe objects in many-sorted algebra’s, using variables, (partial) functions and descriptions (THAT x:A: the unique object satisfying A). Expressions may be undefined, due to the use of partial functions or descriptions involving an assertion that is not uniquely satisfied. Moreover, we have the following constructs:

- `A?t` with the same meaning as THAT x(x = t AND A)
- `s|t` with the same meaning as THAT x(x = s OR x = t)

(assuming x not free in A, s, t). So `A?t` is equal to `t` if `A` holds, otherwise undefined, and `s|t` acts as the join of `s` and `t`, being undefined if both are defined and different.

Assertions are those of many-sorted predicate logic with the expressions as terms; moreover, there is a definedness predicate `t!`, expressing that `t` is defined.
The semantic domains of the expression and the assertion sublanguage consist of linguistic objects: terms and formulae of the logical language MPL. This is a many-sorted version of Scott's E-logic, a two-valued logic with partial functions. This is in contrast with LPF, the three-valued logic underlying VDM (see [3], [5]; the third truth value is undefined). The main reasons for adopting a two-valued logic are simplicity, the desire to adhere to the well-known theory of two-valued logic and the apparent absence of a compelling reason for a third truth value. See [19] for a comparison of LPF and MPL.

3.2.2 Syntax definition

<expression> ::= <object-var> |
    <function-name> '<(' expression-list ',')'> |
    THAT <varsort> <assertion> |
    LET <assignment-list ',',> ; <expression> |
    <assertion> ? <expression> |
    <expression> | <expression> |
    ( <expression> )

<assertion> ::= TRUE |
    FALSE |
    <expression> |
    <expression> = <expression> |
    <predicate-name> '<(' expression-list ',')'> |
    NOT <assertion> |
    <assertion> AND <assertion> |
    <assertion> OR <assertion> |
    <assertion> => <assertion> |
    <assertion> <=> <assertion> |
    FORALL <varsort-list> <assertion> |
    EXISTS <varsort-list> <assertion> |
    LET <assignment-list ',',> ; <assertion> |
    ( <assertion> )

<varsort> ::= <object-var> : <sortname>

<assignment> ::= <object-var> := <expression>

3.2.3 Semantic domain: the logic MPL

The expressions and assertions in COLD-K² correspond with the terms and formulae of MPL, many-sorted predicate logic with partial functions. MPL is the finitary fragment of MPLω (introduced below), which is studied extensively in [16], to which we refer for more information. In Appendix B, we indicate how MPL can be reduced to ordinary predicate logic with equality.

MPL has, for every sort S, an equality predicate =S and a definedness predicate ↓S (postfix notation, so t ↓S means 't is defined', for terms t of sort S). The quantifiers ∀, ∃ range over defined objects only, so we have

\[ t ↓S \iff \exists x : S(x =_S t). \]

For every sort S, there is a constant ↑S denoting the undefined object of sort S, i.e. -(↑S↓S). Free variables range over possibly undefined objects, i.e. we do not have x ↓S for variables x of sort S. As a consequence, we do not necessarily
have \( \exists x : S(x \downarrow s) \), so sorts can be empty. Sort subscripts in \( \downarrow s \) and \( \uparrow s \) are often omitted.

Functions and predicates are strict, i.e. we have

\[
\begin{align*}
    f(t_1, \ldots, t_n) &\downarrow t_1 \land \ldots \land t_n \downarrow \\
    P(t_1, \ldots, t_n) &\to t_1 \land \ldots \land t_n \downarrow
\end{align*}
\]

for all function and predicate symbols. This also holds for the equality predicate, so e.g. \( \uparrow = \uparrow \) is false. We define, for every sort \( S \), an equivalence predicate \( \equiv_S \) which satisfies \( t \equiv_S t \) for all terms of sort \( S \). It is defined as the symmetric closure of the relation \( \equiv_S \) with the intended meaning \( s \) rewrites to \( t \) for \( s \to t \): whenever \( t \) (is defined and) has value \( x \), then \( s \) (is defined and) has the value \( x \). So

\[
\begin{align*}
    s \to t &\ \overset{\text{def}}{=} \forall x : S(t = s x \to s = s x) \\
    s =_S t &\ \overset{\text{def}}{=} s \to s t \land t \to s s
\end{align*}
\]

Observe that \( s \to t \) is equivalent to \( (t \downarrow s = t) \). Here (as in the definition of \( s \to t \)) the logical arrow \( \to \) points from \( t \) to \( s \); this can be paraphrased by saying that the ‘stream of meaning’ is opposite to the rewrite direction.

In \textbf{MPL} we can define terms by description. This is done as follows: if \( A \) is some formula, then \( \iota x : S(A) \) refers either to the unique defined object \( x \) of sort \( S \) satisfying \( A \), or to the undefined object of sort \( S \) if such an \( x \) does not exist or is not uniquely characterised by \( A \). The axiom of the description operator \( \iota \) reads:

\[
\forall y : S(y = \iota x : S(A) \leftrightarrow \forall x : S(A \leftrightarrow x = y)) \quad (y \text{ not free in } A)
\]

Given a signature \( \Sigma \subseteq \text{SORT} \cup \text{FUNC} \cup \text{PRED} \), a model \( M = M(\Sigma) \) consists of a non-empty domains \( S^M \) for every sort \( S \in \Sigma \), total and strict functions \( f^M \) for every \( f \in \Sigma \) and strict predicates \( P^M \) for every \( P \in \Sigma \). Every domain \( S^M \) contains an object \( *_S^M \) which plays the role of undefined object.

The interpretation of terms and formulae in \( M \) w.r.t. an assignment \( a \) (sending variables to objects of the corresponding domain) is defined as follows:

\[
\begin{align*}
    x^M, a &\overset{\text{def}}{=} a(x) \\
    f(M, \ldots, t_n)^M, a &\overset{\text{def}}{=} f(M, \ldots, t_n) \\
    t^M, a &\overset{\text{def}}{=} *^M_S \\
    \iota x : S(A)^M, a &\overset{\text{def}}{=} \text{the unique } d \in S^M - \{*_S^M\} \\
    \text{such that } M, a[x \to d] \models A \\
    \text{if this } d \text{ exists, otherwise } *^M_S.
\end{align*}
\]

\[
\begin{align*}
    M, a \models s = t &\iff s^M, a \neq t^M, a \neq *^M_S \\
    M, a \models (t \downarrow s) &\iff t^M, a \neq *^M_S \\
    M, a \models P(t_1, \ldots, t_n) &\iff t_1^M, a, \ldots, t_n^M, a > \epsilon P^M \\
    M, a \models \neg A &\iff \text{not } M, a \models A \\
    M, a \models A \land B &\iff M, a \models A \text{ and } M, a \models B \\
    M, a \models \forall x : S(A) &\iff \text{for all } d \in S^M - \{*_S^M\}, M, a[x \to d] \models A \\
    M \models A &\iff M, a \models A \text{ for all assignments } a \\
    \Gamma \models A &\iff (M \models B \text{ for all } B \in \Gamma) \implies M \models A \\
    \models A &\iff M \models A \text{ for all models } M
\end{align*}
\]
We list some properties of $\text{mpl}$, referring to [16] for more information (on $\text{mpl}_\omega$, but directly applicable to $\text{mpl}$; see also 3.3.3).

**Soundness and completeness:** we have

\[ \Gamma \vdash A \text{ iff } \Gamma \models A \]

Soundness is proved straightforwardly with induction over the length of derivations, the completeness is proved using semantic tableaux.

**Elimination of descriptions.** It is possible to translate formulae containing descriptions into equivalent formulae without descriptions. This elimination of $\iota$ is accomplished by a mapping $^\iota$ whose behaviour on atomic formulae is suggested by

\[ (\text{Pt}(\iota x. A))^\iota = \exists x. A \land \exists x (A \land \text{Pt}(x)) ; \]

$i$ commutes with the logical operators. We have (see [16, 3.1.3] for more details):

\[
\begin{align*}
\text{mpl} & \vdash A \leftrightarrow A^\iota, \\
\text{mpl} & \vdash A \Leftrightarrow \text{mpl} \vdash (i) \vdash A^\iota.
\end{align*}
\]

**Interpolation.** An interpolant for $A \vdash B$ is a formula $I$ with:

\[
\text{mpl} \vdash (A \rightarrow I) \land (I \rightarrow B) \quad \text{and} \quad \text{par}(I) \subseteq \text{par}(A) \cap \text{par}(B),
\]

where $\text{par}(A)$, the collection of parameters of $A$, is defined as the collection of free variables and signature elements occurring in $A$, including the sorts of all terms occurring in $A$. We have:

If $\text{mpl}: A \vdash B$, then there is an interpolant for $A \vdash B$.

The proof is similar to the proof for interpolation in $\text{mpl}_\omega$, which is given in [16, 3.3]. It is based on the cut-elimination property for $\text{mpl}^-$, i.e. $\text{mpl}^-$ minus the non-logical axioms, and this result is extended to full $\text{mpl}$.

### 3.2.4 Interpretation of the expression language and the assertion language

We now give the interpretation of the expression and the assertion language. First we present a list of variables (without possible subscripts) ranging over different collections of COLD-K$^2$-constructs.

\[
\begin{align*}
S & \text{ sort names} \\
\mathfrak{f} & \text{ function names} \\
P & \text{ predicate names} \\
\mathfrak{x} & \text{ object variables} \\
A & \text{ assertions} \\
B & \text{ expressions}
\end{align*}
\]

We assume a canonical mapping from signature elements and variables of COLD-K$^2$ to $\text{mpl}$, which in our notation corresponds with going from the Courier font (used for COLD) to the Times font (used for $\text{mpl}$). Moreover we assume that $x$ does not occur in $[A]$, $[\mathfrak{x}]$ or $[\mathfrak{f}]$ and that $S$ is the sort of $[\mathfrak{f}]$. 

7
3.3 The definition language

3.3.1 Introduction

The items that are defined in the definition language are sorts, predicates and functions. A definition has in general two aspects, viz. declarative and assertional. A declaration introduces a sort, predicate or function name (the latter two provided with a type): in the assertional part of a definition, the meaning of a sort, predicate and/or function is given directly (by a defining expression or assertion) or indirectly (by an axiom). Looking at the four kinds of definitions, we see:

- sort definitions only have a declarative aspect;
- predicate and function definitions are both declarative and assertional;
- axioms are purely assertional, without a declarative aspect.

The semantics of the declarative aspect of definitions is straightforward. Idem for explicit function and predicate definitions (with keyword DEF) and for axioms. For implicit definitions (keyword IND) the situation is more subtle. The idea is that the meaning of a predicate \( P \) defined by \( \text{IND} \ A \) is: the smallest \( P \) that satisfies \( A \). In order to achieve this in the definition of the semantics it would suffice to extend \( \text{MPL} \) to full second-order logic, but this logic was considered to be far too strong and not well enough understood to provide the semantics for \( \text{COLD} \). Instead the logic \( \text{MPL}_\omega \) has been chosen, an extension of \( \text{MPL} \) with countably infinite conjunctions and disjunctions. \( \text{MPL}_\omega \) has two important properties:

- the Interpolation Property, desirable for an adequate semantics of modularisation and parametrisation, holds for \( \text{MPL}_\omega \);
- fixpoints \( \mu \Phi \) of continuous predicate functions \( \Phi \) can be expressed explicitly in \( \text{MPL}_\omega \) as follows: \( \mu \Phi = \forall n \bigwedge_{n} A_n \) with \( A_0 = \bot, A_{n+1} = \Phi(A_n) \).
In order to make this work, one step has to be taken: transform assertions \( A \) (intended to be inductive definitions) into continuous predicate functions whenever possible. This is worked out in [23]; in 3.3.4 an overview is given.

An alternative to this approach, circumventing this last step, is: restrict the use of inductive definitions in COLD to those explicitly denoted as the least fixpoint of continuous predicate functions. However, the disadvantage of this is that such definitions are more difficult to devise and harder to read.

### 3.3.2 Syntax definition

\[
\begin{align*}
<\text{definition}> : & = \text{SORT} <\text{sort-name}> \\
& \quad \mid \text{FUNC} \ <\text{typed function}> \ [ \ <\text{function body}> ] \\
& \quad \mid \text{PRED} \ <\text{typed predicate}> \ [ \ <\text{predicate body}> ] \\
& \quad \mid \text{AXIOM} \ <\text{assertion}> \\
<\text{typed function}> : & = <\text{function-name}> <\text{function type}> \\
<\text{typed predicate}> : & = <\text{predicate-name}> <\text{predicate type}> \\
<\text{function type}> : & = <\text{sort-name-list} \ ' \#' > \rightarrow <\text{sort-name}> \\
<\text{predicate type}> : & = <\text{sort-name-list} \ ' \#' > \\
<\text{function body}> : & = \text{PAR} <\text{object-var-list}> \ \text{DEF} \ <\text{expression}> \\
& \quad \mid \ \text{IND} \ <\text{assertion}> \\
<\text{predicate body}> : & = \text{PAR} <\text{object-var-list}> \ \text{DEF} \ <\text{assertion}> \\
& \quad \mid \ \text{IND} \ <\text{assertion}>
\end{align*}
\]

### 3.3.3 \( \text{MPL}_w \)

Here we shall show how explicit definitions of a large class of inductively defined predicates and functions can be obtained in the infinitary extension \( \text{MPL}_w \) of \( \text{MPL} \). More information on \( \text{MPL}_w \) is in [16]; the technique used here to make inductive definitions explicit is also described more generally in [23].

\( \text{MPL}_w \) is obtained from \( \text{MPL} \) by adding countably infinite conjunctions \( \bigwedge_n A_n \) (where \( <A_n>_{n<\omega} =<A_0,A_1,\ldots> \) are formulae containing only finitely many different free variables).

Infinite disjunctions are definable: \( \lor_n A_n = \neg \bigwedge_n \neg A_n \). The proof system of \( \text{MPL}_w \) contains the following rules:

\[
\begin{align*}
(\bigwedge L) & \quad \frac{\Gamma, A_i \vdash \Delta}{\Gamma, \bigwedge_n A_n \vdash \Delta} \quad \text{for all } i \\
(\bigwedge R) & \quad \frac{\Gamma \vdash \Delta, A_n >n<\omega}{\Gamma \vdash \Delta, \bigwedge_n A_n}
\end{align*}
\]

\( \text{MPL}_w \) is a conservative extension of \( \text{MPL} \), i.e.

\( \text{MPL}_w \vdash A \iff \text{MPL} \vdash A \) (\( A \) in the language of \( \text{MPL} \)).

\( \iff \) is trivial, \( \Rightarrow \) follows from \( \text{MPL}_w \) : \( \Gamma \vdash \Delta \Rightarrow \text{MPL} \); \( \Gamma \vdash \Delta \) for \( \text{MPL} \)-sequents \( \Gamma \vdash \Delta \), and this is proved by induction over a cut-free derivation of \( \text{MPL}_w \) : \( \Gamma \vdash \Delta \).

\( \text{MPL}_w \) shares the properties mentioned above in 3.2.3 for \( \text{MPL} \): Soundness & completeness, Eliminability of descriptions, Interpolation.
Before we can embark on inductive definitions, we have to introduce some notation.

If \( A \) is a formula, then \( \{x: S_1, \ldots, x_n: S_n \mid A\} \), or \( \{x: S \mid A\} \) or even \( \{x \mid A\} \) for short, is a defined predicate of type \( S = S_1, \ldots, S_n \). The meaning of \( \{x \mid A\} \) is given by

\[
\{x \mid A\}(t) =_{\text{def}} [x := t]A \land t \downarrow
\]

for terms \( t \) of sort \( S \). \( t \downarrow \) has been added in order to make defined predicates strict. Predicate symbols \( P \) of type \( S \) are identified with the defined predicate \( \{x \mid P(x)\} \). Inclusion and extensional equality between defined predicates are defined by

\[
\begin{align*}
\{x \mid A\} \subseteq \{x \mid B\} &=_{\text{def}} \forall x: S(A \rightarrow B),
\{x \mid A\} = \{x \mid B\} &=_{\text{def}} \forall x: S(A \leftrightarrow B),
\{x \mid A\} \equiv \{x \mid B\} &=_{\text{def}} \vdash \{x \mid A\} = \{x \mid B\}.
\end{align*}
\]

We also put

\[
\begin{align*}
S &=_{\text{def}} \{x: S \mid \top\},
\emptyset_S &=_{\text{def}} \{x: S \mid \bot\},
x^e &=_{\text{def}} \{y \mid y \neq x\},
\bigcup \{D_n \mid n \in \omega\} &=_{\text{def}} \{x \mid \bigvee_n A_n\} \text{ if } D_n = \{x \mid A_n\}
\bigcap \{x \mid A(x, y) \mid B(y)\} &=_{\text{def}} \{x \mid \forall y(B(y) \rightarrow A(x, y))\}
\end{align*}
\]

As a consequence of this last definition, we have

\[
\bigcap \emptyset = \bigcap \{X(y) \mid \bot\} = S.
\]

If \( P \) is a predicate symbol and \( D \) a defined predicate of the same type, then \( [P := D] \) is a predicate substitution, defined straightforwardly. The following rule is a derived rule in \( \text{MPL}_\omega \):

\[
\Gamma \vdash \Delta \quad \frac{[P := D]\Gamma \vdash [P := D]\Delta}{(\text{psub})} \quad [P := D]A
\]

We also introduce the predicate substitutions \( [P^+ := D] \), \( [P^- := D] \). They are defined on formulae of \( \text{MPL}_\omega \) not containing descriptions; the intended meaning is that only the positive resp. negative occurrences of \( P \) are replaced by \( D \). The clauses of the definitions of \( [P^+ := D]A \), \( [P^- := D]A \) are similar to those for \( [P := D]A \), with the following modifications:

\[
\begin{align*}
[P^+ := D]P(t) &=_{\text{def}} D(t)
[P^- := D]P(t) &=_{\text{def}} P(t)
[P^+ := D]A \rightarrow A &=_{\text{def}} [P^- := D]A
[P^- := D]A \rightarrow A &=_{\text{def}} [P^+ := D]A
\end{align*}
\]

Observe that \( A \equiv B \) does not imply \( [P^+ := D]A \equiv [P^+ := D]B \); take \( A = (P \rightarrow P), B = (Q \rightarrow Q) \), then \( A \equiv B \) (for \( \vdash A \) and \( \vdash B \)), but \( [P^+ := D]A = P \rightarrow D \) is not equivalent to \( [P^+ := D]B = Q \rightarrow Q \). A similar argument holds for \( [P^- := D] \). To preserve \( \equiv \) under \( [P := D] \), \( \equiv \) has to be strengthened to \( \equiv_p \), strong equivalence in \( P \), defined by

\[
A \equiv_p B =_{\text{def}} [P^+ := Q]A \equiv [P^+ := Q]B,
\]

where \( Q \) is a fresh predicate variable of the same type as \( P \). Now we have

10
\( (S\text{Eq}) \quad A \equiv P \Rightarrow [P^+ := D, P^- := E]A \equiv [P^+ := D, P^- := E]B. \)

If \( P \) is a predicate symbol of type \( S \) and \( A \) is a formula, then \( \Gamma = \Lambda P.A \) is a *predicate function* of type \( S \), with the defining equation

\[
(\Lambda P.A)D =_{\text{def}} [P := D]A
\]

for defined predicates \( D \) of type \( S \).

*Predicate operators* are defined analogously, but with defined predicates instead of formulae: if \( P \) is a predicate symbol of type \( S \) and \( D = \{ x \mid A \} \) is a defined predicate of type \( T \), then \( \Lambda P.D \) is a predicate operator of type \( S \to T \), which satisfies

\[
(\Lambda P.\{ x \mid A \})E =_{\text{def}} \{ x \mid [P := E]A \}
\]

for defined predicates \( E \) of type \( S \). In order to distinguish them, we use \( \Phi \) for predicate functions and \( \Gamma \) for predicate operators.

A predicate operator \( \Gamma \) is called *monotonic* if it satisfies \( D \subseteq E \Rightarrow \Gamma(D) \subseteq \Gamma(E) \). \( \Gamma \) is called *continuous* if it satisfies

\[
D_0 \subseteq D_1 \subseteq D_2 \subseteq \ldots \Rightarrow \Gamma(\cup\{ D_n \mid n \in \omega \}) \equiv \cup\{ \Gamma(D_n) \mid n \in \omega \}
\]

It is clear that continuity implies monotonicity.

\[
\lambda x.t \text{ (or } \lambda x : S. t, \text{ or } \lambda x_1 \ldots x_n : S_1 \ldots S_n. t) \text{ is a *defined function*, with the meaning given by}
\]

\[
s \downarrow \Rightarrow (\lambda x.t)(s) \simeq [x := s]t.
\]

Observe that we do not define

\[
(\lambda x.t)(s) =_{\text{def}} [x := s]t;
\]

this would yield non-strict functions: if \( c \) is a constant, then we would have \( (\lambda x.c)(\bar{t}) = c \). So, generally speaking, we have to consider \( (\lambda x.t)(s) \) as an extension of the language, not as a mere abbreviation. However, if \( t \) has the form \( t y.A(x,y) \), then \( (\lambda x.t)(s) \) can be defined as an abbreviation:

\[
(\lambda x.ty.A(x,y))(s) =_{\text{def}} ty.A(s,y) \land s \downarrow.
\]

Function symbols \( f \) are identified with the defined function \( \lambda x.f(x) \). We also put

\[
\begin{align*}
\lambda x.s & \subseteq \lambda x.t \quad =_{\text{def}} \forall x(s \downarrow \Rightarrow s = t) \\
\lambda x.s & =_{\text{def}} \lambda x.t \quad =_{\text{def}} \forall x(s \simeq t) \\
\lambda x.s & \equiv \lambda x.t \quad =_{\text{def}} \vdash \lambda x.s = \lambda x.t
\end{align*}
\]

A (defined) predicate \( F = F(x,y) \) is called *functional* (notation: \( \text{Func}(F) \)) if

\[
\forall xyz(F(x,y) \land F(x,z) \Rightarrow y = z).
\]
3.3.4 Inductive definition of predicates

An inductively defined predicate $P$ is defined as the smallest predicate satisfying some inductive definition $A(P)$. Let us write $\delta P.A$ for the predicate defined inductively by $A$ (using the predicate parameter $P$), then we have certain conditions (denoted by $A \in \text{Adm}(P)$ and worked out below):

\begin{enumerate}
  \item $[P := \delta P.A]A$, \quad (\delta P.A \text{ satisfies } A);
  \item $A \rightarrow \delta P.A \subseteq P$, \quad (if $P$ satisfies $A$, then $P$ extends $\delta P.A$.)
\end{enumerate}

First we consider the easy case: $A$ happens to be of the form $\Gamma(P) \equiv P$ where $\Gamma$ is continuous. In that case, $\text{Fix}(\Gamma) = \bigcup \{\Gamma^n(\emptyset) \mid n \in \omega\}$ will do for $\delta P.A$, and $\text{Fix}(\Gamma)$ has an explicit definition in $\mathbf{PL}_\omega$ (using an infinite disjunction):

$$\text{Fix}(\Gamma)(t) = \bigvee_n (\Gamma^n(\emptyset))(t).$$

or the general case, we use the technique explained in [23] which we paraphrase here as follows. We put

$$\Delta P.A \overset{\text{def}}{=} \Delta P_\Gamma \{x \mid \neg[P^+ := x^?] A\},$$

$$\delta P.A \overset{\text{def}}{=} \text{Fix}(\Delta P.A).$$

Now, in order to be able to show that this $\delta P.A$ indeed satisfies (I1) and (I2), we have to define the admissibility condition $\text{Adm}(P)$. This is done as follows. A predicate function $\Phi$ is called $\sqcap$-preserving iff, for all $A, B$

$$\forall y (B(y) \rightarrow \Phi(\{x \mid A(x, y)\}) \rightarrow \Phi(\bigcap \{x \mid A(x, y) \mid B(y)\})) ;$$

Now

$$A \in \text{Adm}(P) \text{ iff } \Delta P.A \text{ is continuous } \& \Delta P_\Gamma [P^+ := Q] A$$

is $\sqcap$-preserving

($Q$ is a fresh predicate variable of the same type as $P$). Observe that, for $\sqcap$-preserving $\Phi$, we have $\Gamma(S)$ (for $S = \bigcap \emptyset = \bigcap \{X(y) \mid \bot\}$).

$\text{Adm}(P)$ is not closed under $\equiv$ : consider

$$A \overset{\text{def}}{=} (Pa \lor Pb) \rightarrow (Pa \lor Pb);$$

$A \equiv T, T \in \text{Adm}(P)$, but $A \notin \text{Adm}(P)$, for $\Delta P_\Gamma [P^+ := Q] A$ is not $\sqcap$-preserving: we have (taking $X(y) = \overset{\text{def}}{=} \{z \mid B(y, z)\})$

\begin{align*}
(\Delta P_\Gamma [P^+ := Q] A)(\bigcap \{X(y) \mid T\}) &= (Qa \lor Qb) \\
&\rightarrow (\forall y B(y, a) \lor \forall y B(y, b)) \\
\bigcap (\Delta P_\Gamma [P^+ := Q] A)(\{X(y) \mid T\}) &= \forall y ((Qa \lor Qb) \\
&\rightarrow (B(y, a) \lor B(y, b)))
\end{align*}

and these two are not equivalent. However, $\text{Adm}(P)$ is closed under $\equiv_p$.
Theorem. Let $A \in \text{Adm}(P)$. Then

(i) $\vdash [P := \delta P.A]A$;
(ii) $\vdash A \rightarrow \delta P.A \subseteq P$.

In words: $\delta P.A$ is an explicit definition of the predicate $P$, inductively defined by $A$.

Proof.
(i) Let $\Phi = \text{def } \Lambda P[P^- := \delta P.A]A$. Then $\Phi$ is $\cap$-preserving, for $A \in \text{Adm}(P)$. Now we have

\[ \vdash \forall x (\Phi(x) \rightarrow \Phi(x')) \]
\[ \vdash \Phi(\{ x \mid x = x \rightarrow \Phi(x') \}) \quad \text{(contraposition)} \]
\[ \vdash \Phi(\{ x \mid \neg \Phi(x') \}) \]
\[ \vdash \Phi(\{ x \mid \neg [P^- := \delta P.A, P^+ := x']A \}) \quad \text{(definition of $\Phi$)} \]
\[ \vdash \Phi(\{ x \mid \neg \Phi(x') \}) \quad \text{(definition of $\Delta P.A$)} \]
\[ \vdash [P := \delta P.A]A \quad \text{(definition of $\Phi$)} \]

(ii). We have

\[ \vdash A \land \neg P(x) \rightarrow [P^+ := x']A \quad \text{(monotonicity of $\land$)} \]
\[ \vdash A \leftrightarrow \forall x \{ x \mid [P^+ := x']A \rightarrow P(x) \} \quad \text{(contraposition)} \]
\[ \vdash A \rightarrow \{ x \mid [P^+ := x']A \subseteq P \} \quad \text{(definition of $\subseteq$)} \]
\[ \vdash A \rightarrow (\Delta P.A)P \subseteq P \quad \text{(definition of $\Delta P.A$)} \]
\[ \vdash A \rightarrow \delta P.A \subseteq P^- \quad \text{(by (2))} \]

Observe that only one instance (viz. $\{ x' \mid \Phi(x') \}$) of the $\cap$-preserving property of $\Phi$ has been used.

We establish a syntactically defined subset of $\text{Adm}(P)$. We have

\[ \text{Adm}(P) \mathrel{=} \text{def } \{ A \mid \neg [P^+ := x']A \in Cts(P, x) \land [P^- := Q]A \in \text{Pres}(P) \} \]

where

\[ Cts(P, x) = \text{def } \{ A \mid \Lambda P \{ x \mid A \} \text{ is continuous} \}, \]
\[ \text{Pres}(P) = \text{def } \{ A \mid \Lambda P.A \text{ is } \cap\text{-preserving} \} \]

The following properties of $Cts$ and $\text{Pres}$ are proved easily.

i) $Cts(P, x)$ contains all formulae of the form

\[ \forall \exists (A \land P(\ldots) \land P(\ldots)) \quad (P \text{ not in } A), \]

i.e. all (finite or infinite) disjunctions of existentially quantified formulae $A_1 \land \ldots \land A_n$, where $A_i$ is of the form $P(t_1, \ldots, t_k)$ or does not contain $P$. (Observe that $x$ is not explicitly involved in the definition of this class of formulae.)
ii) \(Pres(P)\) contains all formulae of the form

\[
\forall (A \land t \downarrow P(t)) \quad (P \text{ not negatively in } A),
\]

i.e. all (finite or infinite) conjunctions of universally quantified formulae of the form \(A \land t_1 \downarrow \ldots \land t_n \downarrow \rightarrow P(t_1, \ldots, t_n)\), where \(A\) does not contain \(P\) negatively.

Now we define: a \textit{Horn formula} in \(P\) is a formula of the form

\[
\forall (A \land P(...) \land \ldots \land P(...) t \downarrow \rightarrow P(t)) \land \quad (P \text{ not in } A),
\]

i.e. a (finite or infinite) conjunction of universally quantified formulae of the form \(A_1 \land \ldots \land A_m \land t_1 \downarrow \land \ldots \land t_n \downarrow \rightarrow P(t_1, \ldots, t_n) \quad (m, n \geq 0)\), where \(A_i\) is of the form \(P(t'_1, \ldots, t'_n)\) or does not contain \(P\). The collection of all Horn formulae in \(P\) is denoted \(Horn(p)\). From (i), (ii) it follows that

\[
Horn(P) \subseteq \text{Adm}(P).
\]

In the sequel, we shall allow ourselves some sloppiness in the use of the phrase “\(\ldots \vDash Horn(P)\)”, in the following sense: whenever \(s \downarrow\) always holds in the context under consideration, \(Horn(P)\) is supposed to contain also the formulae of the form

\[
\forall (A \land P(...) \land \ldots \land P(...) t \downarrow \rightarrow P(s, t)),
\]

although such formulae are in fact only \(\equiv P\)-equivalent to elements of \(Horn(P)\), hence in \(\text{Adm}(P)\) by the next lemma. Examples of such terms \(s\) are: bound variables, and the terms \(0\) and \(Sx\) (\(x\) ranging over the natural numbers) in the context of arithmetic.

### 3.3.5 Inductive definition of functions

The explicit definition of an inductively defined function can be obtained as follows: replace the function \(f\) of type \(S_1 \times \ldots \times S_n \rightarrow S_{n+1}\) by an associated predicate \(F\) of type \(\text{true} \times \ldots \times \text{true} \times \text{true} + 1\), give an explicit definition of \(F\) and put \(f(t_1, \ldots, t_n) := \exists x : S_{n+1}. F(t_1, \ldots, t_n, x)\). We describe this in somewhat greater detail.

To replace the function \(f\) by the predicate \(F\), the mapping \(F\) is used. Let \(A\) be a formula containing \(f\); we assume that all occurrences of \(f\) in \(A\) are provided with a unique index. We put

\[
\begin{align*}
\epsilon(t) & =_{\text{def}} T \text{ if } f \text{ not in } t \\
\epsilon(f_i(t)) & =_{\text{def}} F(t^F, x_i) \\
\epsilon(g(t)) & =_{\text{def}} \epsilon(t) = (\epsilon(t_1) \land \ldots \land \epsilon(t_m)) \\
& \quad \text{if } g \text{ different from } f \\
(t^F) & =_{\text{def}} t \text{ if } f \text{ not in } t \\
(f_i(t))^F & =_{\text{def}} x_i \\
(g(t))^F & =_{\text{def}} g(t^F) \text{ if } g \text{ different from } f \\
P(t)^F & =_{\text{def}} P(t) \text{ if } f \text{ not in } t \\
P(t)^F & =_{\text{def}} \exists x (\epsilon(t) \land P(t^F)) \text{ otherwise,}
\end{align*}
\]

14
where \( x \) is the list of variables \( x_i \) occurring in \( \epsilon(t) \).

\( F \) commutes with the logical operators.

This mapping \( F \) is akin to the elimination translation \( ^{\prime} \) of descriptions mentioned in 3.2.3. It is straightforward that

\[
(f \leftrightarrow F) \quad \vdash \forall xy(f(x) = y \leftrightarrow F(x, y)) \to (A \leftrightarrow A^F)
\]

holds. We also put

\[
\delta f.A := \lambda x.y.(\delta F.A^F)(x, y).
\]

\[
\text{Adm}(f) := \{A \mid A^F \in \text{Adm}(F)\}.
\]

\[
A \equiv_f B := \delta F.A \equiv f B^F.
\]

**Theorem.** Let \( A \in \text{Adm}(f) \).

Then

i) \( \vdash \text{Func}(\delta F.A^F) \to [f := \delta f.A]A \);

ii) \( \vdash \text{Func}(\delta F.A^F) \to (A \to \delta f.A \subseteq f) \).

In words: if \( \delta F.A^F \) is a functional predicate, then \( \delta f.A \) is an explicit definition of the function \( f \), inductively defined by \( A \).

iii) If \( A, B \in \text{Adm}(f) \) and \( A \equiv B \), then \( \delta f.A \equiv \delta f.B \).

**Proof.** (i) By \( (f \leftrightarrow F) \) we have

\[
\vdash \forall xy(f(x) = y \leftrightarrow F(x, y)) \to (A \leftrightarrow A^F),
\]

so, applying the substitution \( [F := \delta F.A^F] \) and using

\[
\vdash [F := \delta F.A^F]A^F:
\]

\[
\vdash \forall xy(f(x) = y \leftrightarrow (\delta F.A^F)(x, y)) \to A;
\]

now we apply \([f := \delta f.A] \) and get, by the definition of \( \delta f.A \):

\[
\vdash \forall xy([y.(\delta F.A^F)(x, y) = y \leftrightarrow (\delta F.A^F)(x, y)])
\]

\[
\to [f := \delta f.A]A;
\]

since

\[
\vdash \text{Func}(\delta F.A^F)
\]

\[
\to \forall xy([y.(\delta F.A^F)(x, y) = y \leftrightarrow (\delta F.A^F)(x, y)]),
\]

we now get (i).

(ii). By (i) and the substitution \( [F := \{(x, y) \mid f(x) = y\}] \), we have

\[
\vdash A \leftrightarrow [F := \{(x, y) \mid f(x) = y\}]A^F;
\]

combining this with \( \vdash A^F \to \delta F.A^F \subseteq F \) and the same substitution, we get

\[
\vdash A \to \delta F.A^F \subseteq \{(x, y) \mid f(x) = y\};
\]
with

\[ \vdash \text{Func}(\delta F. A F) \]
\[ \rightarrow (\delta F. A F \subseteq \{(x, y) \mid f(x) = y\}) \leftrightarrow \lambda x.y. (\delta F. A F)(x, y) \subseteq f \]

and the definition of \( \delta f.A \) this yields (ii).

(iii) If \( A, B \in \text{Adm}(f) \) then \( A F, B F \in \text{Adm}(F) \), so \( \delta F. A F \equiv \delta F. B F \) and this implies \( \delta f.A \equiv \delta f.B \). \[\square\]

Before defining the subset \( \text{Horn}(f) \) of \( \text{Adm}(f) \), we consider an example.

**Example: addition.** The usual way of writing down an inductive definition of addition is:

\[ (+) \quad \forall x (x + 0 = x) \land \forall xy (x + Sy = S(x + y)) \]

Transforming this in the format used in this section yields

\[ A =_{\text{def}} \forall x(f(x, 0) = x) \land \forall xy \left(f(x, Sy) = S(f(x, y))\right), \text{ so} \]
\[ A F \equiv F \forall x F(x, 0, x) \land \forall xy \exists uv (F(x, Sy, u) \land F(x, y, v) \land u = Sv) \]
\[ \equiv F \forall x F(x, 0, x) \land \forall xy \exists uv (F(x, Sy, Sv) \land F(x, y, v)) \]

Now this does not help us much, for \( A \notin \text{Adm}(f) \), since \( \Delta F.A F \) is not \( \cap \)-preserving; also

\[ \Delta F. A F = F \forall x \{ (x, y, z) \mid (x = z \land y = 0) \}
\[ \forall \exists u \forall v ((y = Sw \land z = Sv) \lor (y = w \land z = v)) \}
\[ = F \forall x \{ (x, 0, x) \mid x = x \}, \]

and the fixpoint of this constant predicate operator is \( \{(x, 0, x) \mid x = x\} \), which is definitely not the graph of the addition function.

In fact this is not surprising, since \((+)\) is ambiguous as a definition: is \( x + Sy \) defined in terms of \( S \) and \( x + y \), or is \( x + y \) defined as the inverse of \( x + Sy \) under \( S \)? The understanding reader knows that the first reading is meant, but here we have to indicate this explicitly in order to obtain an explicit definition of \( + \). This can be done by presenting the equations as rewrite rules. We therefore recall the notation \( s \leftrightarrow t \) defined in 3.2.3 as an abbreviation of \( \forall x(t = x \rightarrow s = x) \) (suggestively: if \( t \) is defined, then \( s \) is defined in the same way). The defining formula of addition can be written as

\[ B =_{\text{def}} \forall x(f(x, 0) \leftrightarrow x) \land \forall xy \left(f(x, Sy) \leftrightarrow S(f(x, y))\right) \]

and this appears to be an element of \( \text{Adm}(f) \) (by the next lemma); also

\[ B F = \forall x F(x, 0, x) \land \forall xy \exists z (\exists u (F(x, y, u) \land Su = z)) \]
\[ \rightarrow \exists v (F(x, Sy, v) \land v = z)) \]
\[ \equiv F \forall x F(x, 0, x) \land \forall xy (F(x, y, u) \rightarrow F(x, Sy, Su)), \]

so
\[ \Delta F.B^F \equiv \Lambda F \{ (p,q,r) \mid \exists \gamma (p = r \land q = 0) \land \exists \mu, \nu (F(x,y,u) \land p = x \land q = Sy \land r = S\nu) \} = \Lambda F \{ (x,0,x) \mid x = x \} \cup \{ (x,Sy,Sz) \mid F(x,y,z) \} \].

To see that the fixpoint \( \delta F.B^F \) of \( \Delta F.B^F \) is functional, it suffices (using the fact that \( 0 \) is functional) to know that \( \Delta F.B^F \) preserves functionality, i.e. that \( \{ (x,0,x) \mid x = x \} \cup \{ (x,Sy,Sz) \mid F(x,y,z) \} \) is functional whenever \( F \) is. This does not follow from the syntactic form of \( B \), but is based on the properties \( Sx \neq 0 \) and \( Sx = Sy \Rightarrow x = y \) of \( S \).

A Horn formula in \( f \) is a formula of the form

\[
\bigwedge \forall (A \land P_1(f\ldots) \land \ldots P_n(f\ldots) \land s \downarrow f(s) \iff t(f\ldots)) \\
(\text{f not in } A,s)
\]

i.e. a (finite or infinite) conjunction of universally quantified formulae of the form \( A_1 \land \ldots \land A_m \land s_1 \downarrow f(s_1,\ldots,s_n) \iff t \) \((m, n \leq 0)\), where \( A_i \) is an atomic formula or does not contain \( f \), and \( f \) does not occur in the terms \( s_i \). The collection of all Horn formulae in \( f \) is denoted \( \text{Horn}(f) \). We have

\[ \text{Horn}(f) \subseteq \text{Adm}(f). \]

To see this, consider a typical element of \( \text{Horn}(f) : A \overset{\text{def}}{=} \forall y (B \land P(f(t_1) \land t_2) \downarrow f(t_2) \iff g(f(t_2))). \) Now \( A^F = \forall y (B \land \exists z_1 (F(t_1, z_1) \land P(z_1)) \land t_2 \downarrow \forall z_2 (\exists z_3 (F(t_2, z_2, z_3) \land z = g(x_3)) \iff \exists z_2 (F(t_2, t_2) \land z = x_2)) \), so

\[ A^F \equiv F \iff \forall x_1, x_2, y_2 (B \land F(t_1, x_1) \land P(x_1) \land t_2 \downarrow F(t_2, t_2) \land z = g(x_3) \iff F(t_2, z)) \]

this last formula belongs to \( \text{Horn}(F) \), hence to \( \text{Adm}(F) \), so \( A \in \text{Adm}(f) \).

A useful syntactic characterisation of functionality is not easy to find. We confine ourselves to observing that, in the simple case that

\[ A = \forall y (f(x) \iff t(f,x)) , \]

\( A \in \text{Horn}(f) \), and also \( \text{Func}(\delta F.A^F) \) holds; this corresponds to the inductive definition

\[ f(x) \simeq \nu y.(t(f,x) = y) \]

which is considered in [16, 4.8]. In the general case, the functionality of the operator \( \delta F.A^F \) involved depends on (axiomatically presented) properties of functions and predicates occurring in \( A \), as in the example given above.

3.3.6 Interpretation of the definition language

For the definition language, we give two interpretations: \( \llbracket \cdot \rrbracket \) for the declarative and \( \llbracket \cdot \rrbracket \) for the definitional aspect. They will be combined in the definition of the interpretation of classes (3.4.4). Besides those introduced in 3.2.4, we use the following variables:

- \( \text{fbody} \) for function bodies,
- \( \text{pbody} \) for predicate bodies.

17
3.4 The scheme language

3.4.1 Introduction

In COLD, scheme is synonymous with module. Flat schemes consist of a collection of definitions; arbitrary schemes are constructed from flat schemes by applying renamings, imports and exports. Import is a binary symmetric operator here, taking in some sense the union of its arguments. The export operator restricts the (visible) signature of a scheme, making hiding and encapsulation possible.

The semantics of a scheme is a theory in $\text{MPI}_{\mathcal{L}_{\text{S}}}$. This corresponds to the theory semantics of Module Algebra as described in [4]. Some differences between that approach and the one pursued here are:

- renamings $\rho$ in [4] are involutive (i.e. $\rho \circ \rho$ is the identity); here we have no such restrictions;
- the good renaming property, used implicitly in [4], has an explicit formulation here.

3.4.2 Syntax definition

<scheme> ::= <scheme-var>
  | CLASS <definition-list> END
  | RENAME <renaming> IN <scheme>
  | IMPORT <scheme> INTO <scheme>
  | EXPORT <signature> FROM <scheme>
  | LET <scheme-var> := <scheme> ; <scheme>

<renaming> ::= <namepair-list ‘,’>
  | <renaming> $ <renaming>

<namepair> ::= <sort-name> T0 <sort-name>
  | <predicate-name> T0 <predicate-name>
| \(<\text{function-name}>\) T0 \(<\text{function-name}>\)

\(<\text{signature}>\) ::= \(<\text{item-list}>\)
  | \(<\text{renaming}>\) 0 \(<\text{signature}>\)
  | \(<\text{signature}>\) + \(<\text{signature}>\)
  | \(<\text{item}>\) \A \"\signature\"
  | SIG \(<\text{scheme}>\)

\(<\text{item}>\) ::= SORT \(<\text{sort-name}>\)
  | FUNC \(<\text{typed function}>\)
  | PRED \(<\text{typed predicate}>\)

### 3.4.3 Semantic domain: the algebra of theories

The algebra of theories we describe here has three sorts: signatures, theories and renamings. We give a short description of this algebra and its properties, based on [21], where more information and references can be found.

Sorts, typed predicates and typed functions are collectively called signature elements. We assume that, for every type, there are infinitely many signature elements with that type available; this will allow us to apply the fresh signature element principle (see below). A signature is a finite set of signature elements. It is called closed if it contains all sorts (if any) occurring in the types of its elements. The closure \(c(\Sigma)\) of a signature is the least closed signature containing \(\Sigma\). If \(X\) is a (collection of) logical expression(s) then \(SIG(X)\) is the closure of the collection of all signature elements occurring in (elements of) \(X\).

Let \(\Gamma\) be a collection of sentences of \(L\), \(\Sigma\) a signature, then the closure of \(\Gamma\) in \(\Sigma\) is defined by

\[
CL(\Sigma, \Gamma) = \{ A \mid M^{P\Sigma}_L \models A \text{ and } SIG(A) \subseteq c(\Sigma) \}
\]

These closures are called theories. \(TH\) is defined as the collection of all theories of \(M^{P\Sigma}_L\).

The union of two theories is the smallest theory containing them, defined by

\[
T + U = \text{def } CL(SIG(T \cup U), T \cup U).
\]

It is obvious that + is commutative, associative and idempotent.

The restriction of a theory to a signature is the closure of that theory in that signature:

\[
\Sigma \sqcap T = \text{def } CL(\Sigma \cap SIG(T), T).
\]

Renamings are finitely generated mappings defined on expressions of \(L\), changing only signature elements and commuting with taking types (i.e. the type of a renamed signature element is the renaming of the type of that signature element). Observe that we are liberal in the definition of renamings in the sense that they need not be bijective, so e.g. \([P := R, Q := R]\) is a correct renaming (if \(P, Q\) and \(R\) have the same type); this is in contrast to [4], where renamings are bijective and even involutive (i.e. if \(\rho(P) = Q\) then \(\rho(Q) = P\)). We define domain and range of a renaming by
\[\text{dom}(\rho) = \{I \mid \rho(I) \neq I\}\]
\[\text{rg}(\rho) = \{\rho(I) \mid \rho(I) \neq I\}\]

All renamings are finitely generated, so domain and range of a renaming are finite and hence signatures. We shall use some injectivity properties of renamings, defined by

\[\text{inj}(\rho, \Sigma, \Pi) = \text{def} \exists I \exists J. (\rho(I) = \rho(J) \rightarrow I = J)\]
\[\text{inj}(\rho, \Sigma) = \text{def} \text{inj}(\rho, \Sigma, \Sigma)\]
\[\text{inj}(\rho) = \text{def} \text{inj}(\rho, \text{dom}(\rho))\]

We also put:

\[\rho \text{ renames } \Sigma \text{ outside } \Pi \text{ iff }\]
\[\text{dom}(\rho) = \Sigma \cap \Pi, \text{rg}(\rho) \cap (\Sigma \cup \Pi) = \emptyset \text{ and } \text{inj}(\rho, \Sigma).\]

We shall use the \textit{good renaming property}:

for any signatures \(\Sigma, \Pi\) there is a renaming \(\gamma(\Sigma, \Pi)\)
renaming \(\Sigma\) outside \(\Pi\),

which follows directly from the finiteness of signatures and the fresh signature element principle.

The application of renamings on theories is defined by

\[\rho(\mathcal{T}) = \text{def} Cl(\rho(S(\mathcal{T})), \{\rho(A) \mid A \in \mathcal{T}\}).\]

It is clear that \(\rho(\mathcal{T}) = \{\rho(A) \mid A \in \mathcal{T}\}\) if \(\text{inj}(\rho, S(\mathcal{T})))\).

The operators introduced above satisfy several properties, of which the least trivial read as follows. Let \(T, U, V\) range over theories, \(\Sigma\) and \(\Pi\) over signatures, \(\rho\) and \(\sigma\) over renamings.

\[(\square \square) \quad \Sigma \square (\Pi \square T) = (\Sigma \cap \Pi) \square T\]
\[(\rho \square) \quad \text{inj}(\rho, S(\mathcal{T})) \in \Sigma, S(\mathcal{T}) \cup \Sigma) \Rightarrow \rho(\Sigma) \square \mathcal{T} = \rho(\Sigma) \square \rho(\mathcal{T})\]
\[(\square \rho) \quad (\text{dom}(\rho) \cup \text{rg}(\rho)) \cap \Sigma = \emptyset \quad \& \quad \text{inj}(\rho, S(\mathcal{T})) \Rightarrow (\Sigma \square \mathcal{T})\]
\[(\square +) \quad S(\mathcal{T}) \cap S(\mathcal{U}) \subseteq \Sigma \Rightarrow \Sigma \square (\mathcal{T} + \mathcal{U}) = \Sigma \square \mathcal{T} + \Sigma \square \mathcal{U}\]
\[(\gamma) \quad \text{dom}(\gamma(\Sigma, \Pi)) = \Sigma \cap \Pi \quad \& \quad \text{rg}(\gamma(\Sigma, \Pi)) \cap (\Sigma \cup \Pi) = \emptyset \quad \& \quad \text{inj}(\gamma(\Sigma, \Pi), \Sigma)\]

The condition in \((\square \square)\) is required to prevent new identifications of names in \(\rho(\mathcal{T})\) and between names in \(\rho(\Sigma)\) and \(\rho(\mathcal{T})\) not present in \(\rho(\Sigma \square \mathcal{T})\); the condition in \((\square +)\) guarantees that \(\rho\) does not affect names in \(\Sigma \square \mathcal{T}\), neither introduces new identifications in \(\mathcal{T}\).

Most of the properties listed have a straightforward proof; only the proof of \((\square +)\) is more involved and makes essential use of the Interpolation Property of \(\text{MPL}_q\).

Now define the language \(TL\) of \textit{theory terms} by

\[T ::= Th \mid \rho(\mathcal{T}) \mid \Sigma \square \mathcal{T} \mid \mathcal{T} + \mathcal{T},\]

where \(Th\) denotes constants \(Th_1, Th_2 \ldots\) referring to specific theories.
We have the following Normal Form Theorem: any expression of TL is equivalent to an normal form expression

\[ \Sigma \Box (\rho_i(T_1) + \ldots + \rho_n(T_n)). \]

To prove this, it suffices to show that the collection of normal forms contains the constants \( T_i \) and is closed under renamings, \( \Box \) and \( + \). Now \( T_i = S(T_i) \Box \rho_i(T_i) \) which is in normal form, and closure under \( \Box \) follows from (\( \Box \Box \)); closure under renamings and \( + \) follows from

1) for any \( \Sigma, T, \rho \) there is a \( \sigma \) with \( \rho(\Sigma \Box T) = \rho(\Sigma) \Box \sigma(T) \);
2) for any \( \Sigma, \Pi, T, U \), there are \( \rho, \sigma \) with \( \Sigma \Box T + \Pi \Box U = (\Sigma \cup \Pi) \Box (\rho(T) + \sigma(U)) \).

(1) relies on (\( \Box \Box \)) and (\( \rho \Box \)); (2) on (\( \Box \rho \)) and (\( \Box + \)). See [17] for more details.

3.4.4 Interpretation of the scheme language

Let \( a \) be an assignment mapping scheme variables to elements of \( TH \); if \( a \) is such an assignment and \( T e TH \), then \( a[X \rightarrow T] \) is the assignment which sends \( X \) to \( T \) and behaves like a with respect to the other scheme variables.

In the following definition of the semantics of the scheme language, the interpretation of schemes and signatures have an assignment as parameter; the interpretation of renamings and items does not depend on assignments. Besides the variables introduced in 3.2.4 and 3.3.6, we shall use the following variables for \( COLD-K^2 \)-constructs:

\[
\begin{align*}
\mathbf{k} & \quad \text{schemes} \\
X & \quad \text{scheme variables} \\
D & \quad \text{definitions} \\
R & \quad \text{renamings} \\
\Sigma & \quad \text{signatures} \\
S & \quad \text{sort names} \\
I & \quad \text{items} \\
\end{align*}
\]

\[
\begin{align*}
[X][a] & = \text{def} \ a(X) \\
\text{CLASS } D_1 \ldots D_n \ \text{END } [a] & = \text{def} \ a(CH(\{D_1\} \cup \ldots \cup \{D_n\})) \\
\text{RENAME R IN } k[[a] & = \text{def} \ [k][a] \\
\text{IMPORT k INTO } l[[a] & = \text{def} \ [l][a] + [k][a] \\
\text{EXPORT } \Sigma \ \text{FROM } k[[a] & = \text{def} \ [k][a] \Box [k][a] \\
\text{LET } X := k ; 1[[a] & = \text{def} \ [1][a] + [k][a] \\
\text{[I1 TO J1, ..., IN TO Jn][a] & = \text{def} \ [I1][a] := [J1], ..., [In][a] := [Jn] \\
\text{R } \circ S[[a] & = \text{def} \ [R][S][a] \\
\text{[I1, ..., In][a] & = \text{def} \ [I1(a), \ldots, [In][a] \\
\Sigma \oplus \Sigma'[[a] & = \text{def} \ [\Sigma][a] + [\Sigma'][a] \\
\Sigma'[[a] & = \text{def} \ [\Sigma][a] - [\Sigma'][a] \\
\text{SIG } k[[a] & = \text{def} \ SIG([k][a])
\end{align*}
\]
\[ \begin{align*}
\text{SORT } S \ [a] & = \text{S} \\
\text{FUNC } f : S_1 \ldots S_n \rightarrow S \ [a] & = \text{f} \\
\text{PRED } P : S_1 \ldots S_n \ [a] & = \text{P}
\end{align*} \]

3.5 The parametrized scheme language

3.5.1 Introduction

Parametrisation is obtained in \text{COLD} by extending the scheme language with (parametrised) scheme variables, a kind of lambda abstraction (over these variables) and application. This leads to a hierarchy of parametrised schemes of arbitrary finite type. An interesting feature is the kind of lambda abstraction used here: \( \lambda X : A.B(X) \) application of this parametrised scheme to \( C \) (of the right type) only leads to \( B(C) \) if \( C \text{ implements } A \). For flat schemes \( C \) and \( A \), we have that \( C \text{ implements } A \) if and only if the meaning of \( C \) (a theory in \text{MPL}_w) extends the meaning of \( A \). This relation is extended straightforwardly to parametrised schemes of higher type.

As an illustration, think of \( B(X) \) as a specification of a sorting routine for sequences of objects of \( X \) w.r.t. some ordering \( R \) also present in \( X \). It is natural then to restrict the application of a parametrisation of \( A \) to those instances \( C \) where the ordering \( R \) is linear. This can be done by taking for \( A \) the specification of a linear ordering \( R \).

The semantics of the parametrised scheme language is given in a type structure on top of the collection of theories used for the semantics of the scheme language.

3.5.2 Syntax definition

\[
\begin{align*}
\text{<parscheme>} & ::= \text{<parscheme-var>} \\
\text{<scheme>} & | \text{LAMBDA } \text{<parscheme-var>} : \text{<parscheme>} \text{ OF } \text{<parscheme>} \\
& | \text{APPLY } \text{<parscheme>} \text{ TO } \text{<parscheme>} \\
& | \text{LET } \text{<parscheme-var>} ::= \text{<parscheme>} ; \text{<parscheme>}
\end{align*}
\]

3.5.3 Semantic domain: the type structure over the algebra of theories

The interpretation of the scheme language is given in terms of a type structure \( TH^\sigma = \{ TH^\tau \mid \tau \text{ type} \} \) over the collection of theories \( TH = TH^0 \). Here type is the set of Curry types, containing 0 and closed under \( \rightarrow \), and \( TH^{\sigma \rightarrow \tau} \) is defined as the collection of functions from \( TH^\sigma \) to \( TH^\tau \).

For \( \tau \text{ type} \), the ordering \( \leq^\tau \) on \( TH^\tau \) with least element \( \bot^\tau \) is defined by

\[
\begin{align*}
\Phi \leq^\sigma \Psi & \quad \text{iff } \forall T \in TH^\sigma (\Phi(T) \leq^\tau \Psi(T)) \\
\bot^0 & \quad = \text{def } C((\emptyset, \emptyset)} \\
\bot^{\sigma \rightarrow \tau} & \quad = \text{def } \land T\in TH^\sigma. \bot^\tau
\end{align*}
\]

We extend the theory language \( TL \) of 3.4.3 to \( TTL \), \textit{typed theory language}, containing terms \( T^\tau \) of type \( \tau \text{ type} \).
\[
T^0 ::= Th | X^0 | \rho(T^0) | \Sigma | T^0 + T^0 \\
T^x ::= X^x | T^{x \to_T} \\
T^{x \to_T} ::= \lambda X^x \geq T^x.T^x
\]

with the restriction that, in \( \lambda X^x \geq A.B, X^x \) does not occur free in \( A \). Conventions: type subscripts are often omitted, and \( \lambda X.T \) abbreviates \( \lambda X \geq \perp T \).

The meaning of terms of type 0 is as in \( TL \); \( AB \) denotes function application of \( A \) to the argument \( B \), and \( \lambda X \geq A.B \) denotes the function sending \( X \) to \( B = B(X) \) provided \( X \geq A \), otherwise to \( \perp \).

Substitution \( [X := A]B \) is defined with \( \alpha \)-conversion, so

\[
[X := A](\lambda Y \geq B.C) =_{def} \lambda Y \geq [X := A]B.[X := A]C
\]

if \( Y \) not free in \( A \)

\[
=_{def} \lambda Z \geq [X := A][Y := Z]B.[X := A] \quad [Y := Z]C \text{ if } Y \text{ free in } A,
\]

where \( Z \) is a fresh variable not occurring in \( A, B \) or \( C \).

Besides the properties listed above, we have here:

\[
\begin{align*}
(\leq^*) & \quad \leq^* \text{ is a partial order with least element } \perp^* \\
(\leq A) & \quad A \leq B \to AC \leq BC \\
(\leq \lambda) & \quad A \leq B \to (\lambda X \geq A.C) \leq (\lambda X \geq B.C) \\
(\leq 2) & \quad \forall X.(X \geq A \to B \leq C) \to (\lambda X \geq A.B) \leq (\lambda X \geq A.C) \\
(\pi) & \quad (C \geq A \to (\lambda X \geq A.B)C = [X := C]B) \wedge \\
& \quad (C \geq A \vee (\lambda X \geq A.B)C = \perp) \\
(\eta) & \quad \forall X.(X \geq A \vee BX = \perp) \to \lambda X \geq A.BX = B \\
& \quad \text{(provided } X \text{ not free in } B) \\
\end{align*}
\]

Observe that not all terms of higher type denote monotonic functions: \( \lambda X.\lambda Y \geq X.B \) is in general not monotonic, and antimonic if \( X \) does not occur in \( B \).

\( (\pi) \) is the partial version of the \( \beta \)-conversion axiom \( \lambda X.A.B = [X := B].A \) of the \( \lambda \)-calculus, \( (\eta) \) is a variant of the \( \eta \)-conversion axiom \( \lambda X.AX = A \) (\( X \) not free in \( A \)).

**Comparison of \( TTL \) with \( \lambda \pi \)-calculus.**

In [11] several variants of the so-called \( \lambda \pi \)-calculus are introduced. The simplest version, untyped \( \lambda \pi \)-calculus, is just \( \lambda \)-calculus with abstraction terms \( \lambda X \geq A.B \); axiomatics is given in the form of a derivation system with sequents \( \Gamma \vdash A \leq B \), where \( \Gamma \) is a collection of inequalities (\( A = B \) is defined as the conjunction of \( A \leq B \) and \( B \leq A \)). The rule for reduction reads (in our notation)

\[
\Gamma \vdash C \geq A \\
\Gamma \vdash (\lambda X \geq A.B)C = [X := C]B
\]

The corresponding reduction relation \( \to \) is defined by: \( \Gamma \vdash A \to A' \) iff \( \Gamma \vdash B \leq D \) and \( A' \) is obtained from \( A \) by replacing a subformula \( (\lambda X \geq B.C)D \) by \( [X := D]C \). The reflexive and transitive closure \( \to^* \) satisfies the Diamond Property:

\[\phi \text{ if } \Gamma \vdash A \to^* B, A \to^* C, \text{then } \Gamma \vdash B \to^* D, C \to^* D \text{ for some } D.\]

The proof given in [11, 1, 2, 3, 7] follows the lines of P. Martin-Löf’s proof of the corresponding property for the \( \lambda \)-calculus, as given in [2, 3, 2].

23
Applied $\lambda\pi$-calculus is the extension of an algebraic system with preorder with a typed version of $\lambda\pi$-calculus. The corresponding derivation system is obtained by adding all sequents with terms in the algebraic language which are valid in the algebraic system. An instance of an algebraic system considered is class algebra, which is closely related to the theory algebra of Appendix E; the resulting applied $\lambda\pi$-calculus can be compared with $TTL$.

Applied $\lambda\pi$-calculus inherits the Diamond Property of the untyped $\lambda\pi$-calculus, and also satisfies Strong Normalisation:

\[(SN) \quad \text{no term of applied } \lambda\pi\text{-calculus has an infinite reduction path.}\]

As remarked in [11, 4.3.9.8], the easiest way to see this is by an idea first used by Plotkin (in the context of a $\lambda$-typed $\lambda$-calculus): define an embedding of the applied $\lambda\pi$-calculus into the typed $\lambda$-calculus, with the only nontrivial clause \((\lambda X \geq A.B)^* = \lambda Y.(\lambda X.B)^*A^*\); this transfers (infinite) reduction paths from applied $\lambda\pi$-calculus to typed $\lambda$-calculus, and \((SN)\) follows from the strong normalisation property of typed $\lambda$-calculus.

**Reduction in $TTL$.** Let $A, B, C$ be $TTL$-terms of appropriate type. We call the transitions
\[(\lambda X \geq A.B)C \rightarrow [X := C]B \quad (\lambda X \geq A.B)C \rightarrow \perp \quad \lambda X \geq A.BX \rightarrow B\]
reductions of the *redex* (the term at the left hand side) to the *reduct* (the term at the right hand side). If $C \geq A$ holds, then the first reduction is called valid; if $C \geq A$ is false (or $[X := C]B$ equals $\perp$), then the second reduction is valid; the third reduction is valid if $\forall X (X \geq A \lor BX = \perp)$ holds. The relation $\rightarrow$ is extended as follows: if $A \rightarrow A'$, then $AB \rightarrow A'B, BA \rightarrow BA', \lambda X \geq A.B \rightarrow \lambda X \geq A'.B$ and $\lambda X \geq B.A \rightarrow \lambda X \geq B.A'$.

The relation $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$; $A \rightarrow^* B$ is called valid if all intermediate $\rightarrow$-steps are valid. By (π) and (η) it is clear that $A = A'$ holds if $A \rightarrow^* A'$ is a valid reduction.

**Theorem** $TTL$ satisfies \((SN)\) and \((\diamond)\).

**Proof**

If $A_0 \rightarrow A_1 \rightarrow \ldots$ is an infinite reduction path in $TTL$, then $A_0^* \rightarrow A_1^* \rightarrow \ldots$ (where * is Plotkin’s interpretation given above) is an infinite reduction path in typed extensional $\lambda$-calculus; but that theory is strongly normalizing, so we have contradiction and conclude that \((SN)\) holds for $TTL$.

By Newman’s lemma (see [2, 3.1.23]), in the presence of \((SN)\) the diamond property is equivalent to the weak diamond property:

\[(W\diamond) \quad \text{if } A \rightarrow A_1 \text{ and } A \rightarrow A_2, \text{ then } A_1 \rightarrow^* B \text{ and } A_2 \rightarrow^* B \text{ for some } B.\]

The proof of \((W\diamond)\) for $TTL$ consists of a case analysis on $A \rightarrow A_1$ and $A \rightarrow A_2$, based on the following observation: if $A \rightarrow A'$ is valid, then one of the following holds (here $\equiv$ means literal identity):

\[\text{if } A \rightarrow A' \text{ is valid, then one of the following holds (here $\equiv$ means literal identity):}\]

\[\text{(W\diamond)} \quad \text{if } A \rightarrow A_1 \text{ and } A \rightarrow A_2, \text{ then } A_1 \rightarrow^* B \text{ and } A_2 \rightarrow^* B \text{ for some } B.\]
We consider a typical case:

\[(5,2): \lambda X \geq B.C) D, A_1 \equiv [X := D] C, D \geq B, A_2 \equiv \lambda X \geq B.C) D', D \rightarrow D'. \]

Take \(A_3 := [X := D'] C\), then \(A_1 \rightarrow A_3\) and \(A_2 \rightarrow A_3\), for \(D' \geq B\) (since \(D' = D \geq B\)). The other cases are treated likewise. □

A term \(N\) of \(TTL\) is a normal form if \(N\) cannot be reduced, i.e. there is no \(N'\) with \(N \rightarrow N'\). As a direct consequence of \((SN)\) and \((\delta)\), we have:

\[
\forall X (X \geq B \land CX = \bot)
\]

every term in \(TTL\) reduces to a unique normal form.

It is not the case that a normal form contains no reducts, even with the restriction to closed normal forms. Consider

\[
\lambda X.\lambda Y.(\lambda Z \geq X.A)Y;
\]

this is a closed term containing the reduct \((\lambda Z \geq X.A)Y\) which cannot be reduced, since \(Y \geq X\) can be either true or false.

### 3.5.4 Interpretation of the parametrized scheme language

Now let \(a\) be an assignment mapping scheme variables to elements of \(TH^w\) of corresponding type; if \(a\) is such an assignment and \(TeTH^w\) has the same type as \(X\), then \(a[X \rightarrow T]\) is the assignment which sends \(X\) to \(T\) and behaves like \(a\) on the other scheme variables. We use the following variables for \(COLD-K^2\)-constructs:

\[
\begin{align*}
K \& \ M & \text{ parschemes} \\
Y & \text{ parscheme variables}
\end{align*}
\]

We define

\[
\begin{align*}
\llbracket Y \rrbracket(a) & \overset{\text{def}}{=} a(Y) \\
\llbracket\Lambda\lambda Y : K\ \llbracket L\rrbracket(a) & \overset{\text{def}}{=} \lambda T \rightarrow \llbracket K \rrbracket(a)[\llbracket L \rrbracket(a)[Y \rightarrow T]] \\
\llbracket\text{APPLY} K \text{ TO } L \rrbracket(a) & \overset{\text{def}}{=} \llbracket K \rrbracket(a)[\llbracket L \rrbracket(a)] \\
\llbracket\text{LET } Y := K \text{ ; } L \rrbracket(a) & \overset{\text{def}}{=} \llbracket L \rrbracket(a)[Y \rightarrow \llbracket K \rrbracket(a)]
\end{align*}
\]

### 3.6 Designs

#### 3.6.1 Introduction

Designs are descriptions of systems consisting of parametrised schemes, using components. Components have a name, a specification and (possibly) an implementation. The semantics is straightforward. See [8] for more information of designs.
3.6.2 Syntax definition

<design> ::= DESIGN <component-list ',' > SYSTEM
            <parscheme-list ',' >

<component> ::= COMP <parscheme-var> : <parscheme>
             != <parscheme>]] LET <parscheme-var> ::= <parscheme>

3.6.3 Interpretation of the design language

Assignments are defined as in 3.5.4. for the definition of the semantics of the
parametrized scheme language. To the variables introduced in 3.5.4 we add $C$
(possibly with subscript) ranging over components.

\[
\begin{align*}
[\text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M_1, \ldots, M_n](a) &= \text{def} \\
&\begin{cases}
[\text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M_1](a), \\
\ldots, \\
[\text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M_n](a)
\end{cases} > \\
[\text{DESIGN SYSTEM } M](a) &= \text{def} [M](a) \\
[\text{DESIGN } Y : K := L; C_1; \ldots; C_m \text{ SYSTEM } M](a) &= \text{def} \\
&\begin{cases}
(a^T \geq [K](a), \text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M] \\
(a[Y \rightarrow T]))(\text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M](a[Y \rightarrow T])
\end{cases} \\
[\text{DESIGN LET } Y : K; C_1; \ldots; C_m \text{ SYSTEM } M] &= \text{def} \\
&\begin{cases}
[\text{DESIGN } C_1; \ldots; C_m \text{ SYSTEM } M](a)[Y \rightarrow [K](a)]
\end{cases}
\end{align*}
\]

References


(eds.) VDM'91: Formal Software Development Methods, LNCS551, Springer-Verlag, 279 - 308


[23] G.R. Renardel de Lavalette, From implicit via inductive to explicit definitions, this volume.

Appendix A: well-formedness

In the previous sections only the context-free part of the syntax definition has been given, which is to be supplemented with well-formedness conditions in order to yield a complete language definition. We do not attempt to give a complete and rigorous definition of well-formedness here, but confine ourselves to presenting an incomplete and informal list of well-formedness conditions.

i) In \text{LET } x_1 \coloneqq t_1, \ldots, x_n \coloneqq t_n, \text{ the object variables } x_1, \ldots, x_n \text{ are mutually different.}

ii) In the definitions

\begin{itemize}
  \item \text{FUNC } f : S_1 \# \ldots \# S_m \rightarrow S \text{ PAR } x_1 \ldots x_n \text{ DEF } t
  \item \text{FUNC } f : S_1 \# \ldots \# S_m \rightarrow S \text{ PAR } x_1 \ldots x_n \text{ IND } A_1
  \item \text{PRED } P : S_1 \# \ldots \# S_m \text{ PAR } x_1 \ldots x_n \text{ DEF } A_2
  \item \text{PRED } P : S_1 \# \ldots \# S_m \text{ PAR } x_1 \ldots x_n \text{ IND } A_3
  \item \text{AXIOM } A_4
\end{itemize}

we have:

\begin{itemize}
  \item \text{m equals } n \text{ and the variables } x_1 \ldots x_n \text{ are all different;}
  \item \text{the free variables in } t, A_1, A_2, A_3 \text{ are among } x_1 \ldots x_n;
  \item \text{A}_4 \text{ is closed, i.e. does not contain free variables;}
  \item \text{f does not occur in } t, \text{ and } P \text{ not in } A_2.
\end{itemize}

iii) For every function symbol \text{f} occurring in \text{CLASS } D_1 \ldots D_n \text{ END, there is exactly one } i \ (1 \leq i \leq n) \text{ such that } D_i \text{ equals FUNC } f : S_1 \# \ldots \# S_m \rightarrow S \text{ body, where } f \text{ body may be empty; this } D_i \text{ is called the definition of } f; \text{ the same holds for predicate symbols.}

iv) all occurrences of function and predicate symbols are correct w.r.t. the type assigned to that symbol by its definition.

v) Renamings of the form

\[ I_1 \text{ TO } J_1, \ldots, I_n \text{ TO } J_n \]

are well-formed if:

\begin{itemize}
  \item \text{the items } I_1, \ldots, I_n \text{ are all different;}
  \item \text{if } I_i \ (1 \leq i \leq n) \text{ denotes a sort, then so does } J_i;
  \item \text{if } I_i \ (1 \leq i \leq n) \text{ equals FUNC } f : S_1 \# \ldots \# S_m \rightarrow S_{m+1}, \text{ then } J_i \text{ equals FUNC } g : S'_1 \# \ldots \# S'_m \rightarrow S'_{m+1}, \text{ satisfying, for all } k \text{ between } 1 \text{ and } m + 1 \text{: } S_k \text{ equals } S'_k \text{ unless for some } l \ (1 \leq l \leq n) J_l \text{ equals SORT } S_k \text{ and } J_i \text{ equals SORT } S'_k;
  \item \text{analogously for predicates.}
\end{itemize}

vi) In \text{LAMBDA } Y : K \text{ OF } L \text{ the scheme variable } Y \text{ does not occur free in } K.
Appendix B: reduction of MPL to first-order logic

We indicate briefly how to reduce MPL to L, first-order logic with equality. It suffices to do so for L- (i).

Signatures for L consist of finite subsets of \( \text{SORT} \cup \text{FUNC} \cup \text{PRED} \{ \uparrow \} \); their elements no longer have a type but do have an arity, indicating the number of arguments. Moreover, the elements of \( \text{SORT} \) are unary predicates and \( \uparrow \) is a constant (i.e. a function with arity 0). The mapping \( L : \text{MPL-} (i) \rightarrow L \) is defined by

\[
\begin{align*}
(\uparrow s)^L &= \text{def} \uparrow^

(s =_t t)^L &= \text{def} S(s) \land s = t

(t \downarrow s)^L &= \text{def} S(t)

(P(t))^L &= \text{def} P(t^L) \text{ for all other predicates } P

(\neg A)^L &= \text{def} \neg A^L

(A \land B)^L &= \text{def} A^L \land B^L

(\forall x : S(A))^L &= \text{def} \forall x(S(x) \rightarrow A^L)
\end{align*}
\]

Abbreviating \( S_1(x_1) \land \ldots \land S_n(x_n) \) by \( S(x) \), we also put

\[
\begin{align*}
Ax(S) &= \text{def} \neg S(\uparrow) \\
Ax(f) &= \text{def} \forall x(S \langle f(x) \rangle \rightarrow S(x)) \\
& \quad \text{if type}(f) = S_1 \times \ldots \times S_n \rightarrow S \\
Ax(P) &= \text{def} \forall x(P(x) \rightarrow S(x)) \\
& \quad \text{if type}(P) = S_1 \times \ldots \times S_n \\
Ax(\Sigma) &= \text{def} \{ Ax(I) \mid I \in \Sigma \} \quad \text{for MPL-signatures } \Sigma.
\end{align*}
\]

Now we have

\[
\text{MPL}(\Sigma) - (i) \vdash A \iff L(\Sigma \cup \{ \uparrow \}) : Ax(\Sigma) \vdash A^L
\]

\( \Rightarrow \) is proved by induction over the length of a derivation of \( \vdash A \). For \( \Leftarrow \), it suffices to see that for every \( M = M(\Sigma) \) there is a model \( M' \) of \( L \) with signature \( \Sigma \cup \{ \uparrow \} \) such that \( M' \models Ax(\Sigma) \) holds, and also \( M' \models A^L \Rightarrow M \models A \); then the result follows with the completeness theorem for MPL.

We indicate with an example how \( M' \) can be obtained from \( M \). Let \( \Sigma = \{ S, S', f : S \rightarrow S', P : S \} \), then \( M = \{ S^M, S'^M, f^M, P^M \} \); now \( M_A = \{ D, f, S, S', P, \uparrow \} \) is given by

\[
\begin{align*}
D &= \text{def} \; S \cup S' \cup \{ \uparrow \} \\
f &= \text{def} \; f^M \cup \{ \langle d, \uparrow \rangle \} \; | \; \{ d \in S \cup S' \} \\
S &= \text{def} \; S^M - \{ *S'^M \} \\
S' &= \text{def} \; S'^M - \{ *S^M \} \\
P &= \text{def} \; P^M.
\end{align*}
\]

30