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_ω (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 Eindhoven (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 relation.

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 Loe 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 [15]. 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 specification 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 betwen 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-K^2 , 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-K^2 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 MPL$_\omega$ (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 $\lambda\pi$ (for the semantics of parametrisation).
- [16] about the logic MPL$_\omega$, 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-K^2

3.1 The description of syntax and semantics

The overall structure of COLD-K^2 can be rendered as follows: the terms refer to sublanguages.

```
                DESIGN
                  |
        PARAMETRIZED 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-K^2 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:

```
<xxxxx-list> : :                = [<xxxxx>]
<xxxxx-list '*' > : :           = [<xxxxx> {*<xxxxx>}]
                                where * is some symbol acting as delimiter
<'*'xxxxx-list '**' '***' > : : = [*< xxxxx> {**<xxxxx>}***]
<xxxx-name> : :                 = <identifier>
<xxxx-var> : :                  = <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 xA: 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-K^2 correspond with the terms and formulae of MPL, many-sorted predicate logic with partial functions. MPL is the finitary fragment of MPL$_\omega$ (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 $\downarrow_S$ (postfix notation, so $t \downarrow_S$ means 't is defined', for terms t of sort S). The quantifiers $\forall, \exists$ range over defined objects only, so we have

$$t \downarrow_S \leftrightarrow \exists x : S(x =_S t).$$

For every sort S, there is a constant $\uparrow_S$ denoting the undefined object of sort S, i.e. $\neg(\uparrow_S \downarrow_S)$. Free variables range over possibly undefined objects, i.e. we do not have $x \downarrow_S$ 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 $=_S, \downarrow_S$ and $\uparrow_S$ are often omitted.

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

$$f(t_1, \ldots, t_n) \downarrow \to t_1 \downarrow \wedge \ldots \wedge t_n \downarrow$$
$$P(t_1, \ldots, t_n) \to t_1 \downarrow \wedge \ldots \wedge t_n \downarrow$$

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 $\simeq_S$ which satisfies $t \simeq_S t$ for *all* terms of sort S. It is defined as the symmetric closure of the relation $\hookrightarrow_S$ with the intended meaning *s rewrites to t* for $s \hookrightarrow t$: whenever t (is defined and) has value x, then s (is defined and) has the value x. So

$$s \hookrightarrow_S t \quad =_{\text{def}} \forall x : S(t =_S x \to s =_S x)$$
$$s \simeq_S t \quad =_{\text{def}} s \hookrightarrow_S t \wedge t \hookrightarrow_S s$$

Observe that $s \hookrightarrow t$ is equivalent to $(t \downarrow\to s = t)$. Here (as in the definition of $s \hookrightarrow t$) 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 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 ι reads:

$$\forall y : S(y = \iota x : S(A) \leftrightarrow \forall x : S\big(A \leftrightarrow x = y\big)) \qquad (y \text{ not free in } A)$$

Given a signature $\Sigma \subseteq$ SORT $\cup$ FUNC $\cup$ PRED, a model $M = M(\Sigma)$ consists of a non-empty domains S^M for every sort $S \epsilon \Sigma$, total and strict functions f^M for every $f \epsilon \Sigma$ and strict predicates P^M for every $P \epsilon \Sigma$. 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:

$$\begin{array}{ll}
x^{M,a} & =_{\text{def}} a(x) \\
f(t_1, \ldots, t_n)^{M,a} & =_{\text{def}} f^M(t_1^{M,a}, \ldots, t_n^{M,a}) \\
\uparrow S^{M,a} & =_{\text{def}} *_S^M \\
\iota x : S(A)^{M,a} & =_{\text{def}} \textit{the unique } d \epsilon S^M - \{{}^*S^M\} \\
 & \qquad \textit{such that } M, a[x \to d] \models A \\
 & \qquad \textit{if this } d \textit{ exists, otherwise } *_S^M.
\end{array}$$

$$\begin{array}{ll}
M, a \models s = t & \textit{iff } s^{M,a} \neq t^{M,a} \neq *S^M \\
M, a \models (t \downarrow_S) & \textit{iff } t^{M,a} \neq *S_M \\
M, a \models P(t_1, ..., t_n) & \textit{iff } < t_1^{M,a}, \ldots, t_n^{M,a} > \epsilon P^M \\
M, a \models \neg A & \textit{iff not } M, a \models A \\
M, a \models A \wedge B & \textit{iff } M, a \models A \textit{ and } M, a \models B \\
M, a \models \forall x : S(A) & \textit{iff for all } d \epsilon S^M - \{{}^*S^M\} M, a[x \to d] \models A \\
M \models A & \textit{iff } M, a \models A \textit{ for all assignments } a \\
\Gamma \models A & \textit{iff } (M \models B \textit{ for all } B \epsilon \Gamma) \textit{ implies } M \models A \\
\models A & \textit{iff } M \models A \textit{ for all models } M
\end{array}$$

We list some properties of MPL, referring to [16] for more information (on MPL_ω, but directly applicable to 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 ι is accomplished by a mapping $^\iota$ whose behaviour on atomic formulae is suggested by

$$\big(Pt(\iota x.A)\big)^\iota = \exists! x A \wedge \exists x(A \wedge Pt(x)); \qquad (\iota)$$

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

$$\text{MPL} \vdash A \leftrightarrow A^\iota,$$
$$\text{MPL} \vdash A \Leftrightarrow \text{MPL} - (\iota) \vdash A^\iota.$$

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

$$\text{MPL} \vdash (A \to I) \wedge (I \to B) \quad \text{and} \quad \text{par}(I) \subseteq \text{par}(A) \cap \text{par}(B),$$

where 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 MPL: $A \vdash B$, then there is an interpolant for $A \vdash B$.

The proof is similar to the proof for interpolation in MPL_ω which is given in [16, 3.3]. It is based on the cut-elimination property for MPL^-, i.e. MPL^- minus the non-logical axioms, and this result is extended to full 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.

`S`	sort names
`f`	function names
`P`	predicate names
`x`	object variables
`A B`	assertions
`s t`	expressions

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

$[\![$ x $]\!]$	$=_{\text{def}}\ x$
$[\![$ f(t1,...,t n)$]\!]$	$=_{\text{def}}\ f([\![$ t1$]\!],\ldots,[\![$tn $]\!])$
$[\![$THAT x:S A $]\!]$	$=_{\text{def}}\ \iota x : S[\![A]\!]$
$[\![$ LET x1:=t1,...,xn:=tn; t$]\!]$	$=_{\text{def}}\ [x_1 := [\![$t1$]\!],\ldots,x_n := [\![tn]\!]][\![t]\!]$
$[\![$A ? t$]\!]$	$=_{\text{def}}\ \iota x : S([\![A]\!] \wedge x = [\![t]\!])$
$[\![$s | t$]\!]$	$=_{\text{def}}\ \iota x : S(x = [\![s]\!] \vee x = [\![t]\!])$
$[\![$(t)$]\!]$	$=_{\text{def}}\ [\![$t$]\!]$
$[\![$ TRUE$]\!]$	$=_{\text{def}}\ \top$
$[\![$ FALSE$]\!]$	$=_{\text{def}}\ \bot$
$[\![$t!$]\!]$	$=_{\text{def}}\ [\![$t$]\!]\downarrow$
$[\![$s = t$]\!]$	$=_{\text{def}}\ [\![$s$]\!] = [\![$t$]\!]$
$[\![$P(t1,...,tn)$]\!]$	$=_{\text{def}}\ P([\![$t1$]\!],\ldots,[\![$tn$]\!])$
$[\![$ NOT A$]\!]$	$=_{\text{def}}\ \neg[\![$A$]\!]$
$[\![$ A AND B$]\!]$	$=_{\text{def}}\ [\![$A$]\!] \wedge [\![$B$]\!]$
$[\![$A OR B$]\!]$	$=_{\text{def}}\ [\![$A$]\!] \vee [\![$B$]\!]$
$[\![$A => B$]\!]$	$=_{\text{def}}\ [\![$A$]\!] \rightarrow [\![$B$]\!]$
$[\![$A <=> B$]\!]$	$=_{\text{def}}\ [\![$A$]\!] \leftrightarrow [\![B]\!]$
$[\![$FORALL x1: S1... xn: Sn A$]\!]$	$=_{\text{def}}\ \forall x_1 : S_1 \ldots \forall x_n : S_n[\![A]\!]$
$[\![$EXISTS x1: S1... xn:Sn A$]\!]$	$=_{\text{def}}\ \exists x_1 : S_1 \ldots \exists x_n : S_n[\![A]\!]$
$[\![$LET x1: = t1,..., xn: =tn; A$]\!]$	$=_{\text{def}}\ [x_1 := [\![$t1$]\!],\ldots,x_n := [\![tn]\!]][\![A]\!]$
$[\![$(A)$]\!]$	$=_{\text{def}}\ [\![$A$]\!]$

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 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 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 COLD. Instead the logic MPL_ω has been chosen, an extension of MPL with countably infinite conjunctions and disjunctions. MPL_ω has two important properties:

- the Interpolation Property, desirable for an adequate sementics of modularisation and parametrisation, holds for MPL_ω;
- fixpoints $\mu\Phi$ of continuous predicate functions Φ can be expressed explicitly in MPL_ω as follows: $\mu\Phi = \vee_n A_n$ 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

<definition> : : = `SORT` <sort-name>
| `FUNC` <typed function> [<function body>]
| `PRED` <typed predicate> [<predicate body>]
| `AXIOM` <assertion>

<typed function> : : = <function-name> <function type>

<typed predicate> : : = <predicate-name> <predicate type>

<function type> : : = <sort-name-list `'#'`> `->` <sort-name>

<predicate type> : : = <sort-name-list `'#'`>

<function body> : : = `PAR` <object-var-list> `DEF` <expression>
| `IND` <assertion>

<predicate body> : : = `PAR` <object-var-list> `DEF` <assertion>
| `IND` <assertion>

3.3.3 MPL$_\omega$

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

MPL$_\omega$ is obtained from 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: $\bigvee_n A_n = \neg \bigwedge_n \neg A_n$. The proof system of MPL$_\omega$ contains the following rules:

$$(\bigwedge L) \quad \frac{\Gamma, A_i \vdash \Delta}{\Gamma, \bigwedge_n A_n \vdash \Delta} \quad \text{for all } i \qquad (\bigwedge R) \quad \frac{< \Gamma \vdash \Delta, A_n >_{n<\omega}}{\Gamma \vdash \Delta, \bigwedge_n A_n}$$

MPL$_\omega$ is a conservative extension of MPL, i.e.

$$\text{MPL}_\omega \vdash A \Leftrightarrow \text{MPL} \vdash A \text{ (A in the language of MPL).}$$

$\Leftarrow$ is trivial, $\Rightarrow$ follows from MPL$_\omega$: $\Gamma \vdash \Delta \Rightarrow$ MPL: $\Gamma \vdash \Delta$ for MPL-sequents $\Gamma \vdash \Delta$, and this is proved by induction over a cut-free derivation of MPL$_\omega$: $\Gamma \vdash \Delta$.

MPL$_\omega$ shares the properties mentioned above in 3.2.3 for 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 $\{x_1 : S_1, \ldots, x_n : S_n \mid A\}$, or $\{\mathbf{x} : \mathbf{S} \mid A\}$ or even $\{\mathbf{x} \mid A\}$ for short, is a *defined predicate* of type $\mathbf{S} = S_1, \ldots, S_n$. The meaning of $\{\mathbf{x} \mid A\}$ is given by

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

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

$$\begin{array}{lll} \{x \mid A\} \subseteq \{x \mid B\} & =_{\text{def}} & \forall x : S(A \to B), \\ \{x \mid A\} = \{x \mid B\} & =_{\text{def}} & \forall x : S(A \leftrightarrow B), \\ \{x \mid A\} \equiv \{x \mid B\} & =_{\text{def}} & \vdash \{x \mid A\} = \{x \mid B\}. \end{array}$$

We also put

$$\begin{array}{lll} S & =_{\text{def}} & \{x : S \mid \top\} \\ \emptyset_S & =_{\text{def}} & \{x : S \mid \bot\} \\ x^c & =_{\text{def}} & \{y \mid y \neq x\} \\ \bigcup\{D_n \mid n \epsilon \omega\} & =_{\text{def}} & \{x \mid \bigvee_n A_n\} \text{ if } D_n = \{x \mid A_n\} \\ \bigcap\{\{x \mid A(x,y)\} \mid B(y)\} & =_{\text{def}} & \{x \mid \forall y(B(y) \to A(x,y))\} \end{array}$$

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 MPL_ω:

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

We also introduce the predicate substitutions $[P^+ := D]$, $[P^- := D]$. They are defined on formulae of MPL_ω 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:

$$\begin{array}{lll} [P^+ := D]P(t) & =_{\text{def}} & D(t) \\ [P^- := D]P(t) & =_{\text{def}} & P(t) \\ [P^+ := D]\neg A & =_{\text{def}} & [P^- := D]A \\ [P^- := D]\neg A & =_{\text{def}} & [P^+ := D]A \end{array}$$

Observe that $A \equiv B$ does *not* imply $[P^+ := D]A \equiv [P^+ := D]B$: take $A = (P \to P), B = (Q \to Q)$, then $A \equiv B$ (for $\vdash A$ and $\vdash B$), but $[P^+ := D]A = P \to D$ is not equivalent to $[P^+ := D]B = Q \to Q$. 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

$(SEq) \quad A \equiv_P B \Rightarrow [P^+ := D, P^- := E]A \equiv [P^+ := D, P^- := E]B.$

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 Φ for predicate functions and Γ for predicate operators.

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

$$D_0 \subseteq D_1 \subseteq D_2 \subseteq \ldots \Rightarrow \Gamma(\cup\{D_n \mid n\epsilon\omega\}) \equiv \cup\{\Gamma(D_n) \mid n\epsilon\omega\}$$

It is clear that continuity implies monotonicity.

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

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

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 $(\lambda x.c)(\uparrow) = c$. 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 $\iota y.A(x,y)$, then $(\lambda x.t)(s)$ can be defined as an abbreviation:

$$\big(\lambda x.\iota y.A(x,y)\big)(s) =_{\text{def}} \iota y.\big(A(s,y) \wedge s \downarrow\big).$$

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

$$\begin{array}{ll} \lambda x.s \subseteq \lambda x.t & =_{\text{def}} \forall x(s \downarrow\to s = t) \\ \lambda x.s = \lambda x.t & =_{\text{def}} \forall x(s \simeq t) \\ \lambda x.s \equiv \lambda x.t & =_{\text{def}} \vdash \lambda x.s = \lambda x.t \end{array}$$

A (defined) predicate $F = F(x,y)$ is called *functional* (notation: Func(F)) if

$$\forall xyz\big(F(x,y) \wedge F(x,z) \to y = z\big).$$

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 under certain conditions (denoted by $A \epsilon \mathrm{Adm}(P)$ and worked out below):

$$\begin{array}{lll} \text{(I1)} & [P := \delta P.A]A, & (\delta P.A \text{ satisfies } A); \\ \text{(I2)} & A \to \delta P.A \subseteq P, & (\text{if } P \text{ satisfies } A, \text{ then } P \text{ extends } \delta P.A.) \end{array}$$

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

$$Fix(\Gamma)(t) = \bigvee_n \big(\Gamma^n(\emptyset)\big)(t).$$

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

$$\begin{array}{ll} \Delta P.A & =_{\mathrm{def}} \Lambda P.\{x \mid \neg[P^+ := x^c]A\}, \\ \delta P.A & =_{\mathrm{def}} Fix(\Delta P.A). \end{array}$$

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 $\mathrm{Adm}(P)$. This is done as follows. A predicate function Φ is called $\bigcap$-*preserving* iff, for all A, B

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

Now

$$\begin{array}{l} A \epsilon\ \mathrm{Adm}(P) \text{ iff } \Delta P.A \text{ is continuous } \&\ \Lambda P.[P^- := Q]A \\ \quad \text{is } \bigcap\text{-preserving} \end{array}$$

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

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

$$A =_{\mathrm{def}} (Pa \vee Pb) \to (Pa \vee Pb);$$

$A \equiv T, T \epsilon \mathrm{Adm}(P)$, but $A \notin \mathrm{Adm}(P)$, for $\Lambda P.[P^- := Q]A$ is not $\bigcap$-preserving: we have (taking $X(y) =_{\mathrm{def}} \{z \mid B(y,z)\}$)

$$\begin{array}{ll} (\Lambda P.[P^- := Q]A)\ (\cap\{X(y) \mid T\}) & = (Qa \vee Qb) \\ & \quad \to \big(\forall y B(y,a) \vee \forall y B(y,b)\big) \\ \bigcap(\Lambda P.[P^- := Q]A)\{X(y) \mid T\} & = \forall y\Big((Qa \vee Qb) \\ & \quad \to \big(B(y,a) \vee B(y,b)\big)\Big) \end{array}$$

and these two are not equivalent. However, $\mathrm{Adm}(P)$ is closed under $\equiv_P$.

Theorem Let $A\epsilon\mathrm{Adm}(P)$. Then
i) $\vdash [P := \delta P.A]A$;
ii) $\vdash A \to \delta P.A \subseteq P$.
In words: $\delta P.A$ is an explicit definition of the predicate P, inductively defined by A.

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

$$\begin{array}{ll}
\vdash \forall x\big(\Phi(x^c) \to \Phi(x^c)\big) & \\
\vdash \Phi\big(\cap\{x^c \mid \Phi(x^c)\}\big) & (\Phi \text{ preserves } \bigcap) \\
\vdash \Phi\Big(\{y \mid \forall x\big(\Phi(x^c) \to y \neq x\big)\}\Big) & \\
\vdash \Phi\Big(\{y \mid \forall x\big(y = x \to \neg\Phi(x^c)\big)\}\Big) & (\text{contraposition}) \\
\vdash \Phi\big(\{x \mid \neg\Phi(x^c)\}\big) & \\
\vdash \Phi(\{x \mid \neg[P^- := \delta P.A, P^+ := x^c]A\}) & (\text{definition of } \Phi) \\
\vdash \Phi\big((\Delta P.A)(\delta P.A)\big) & (\text{definition of } \Delta P.A) \\
\vdash \Phi(\delta P.A) & (\text{continuity of } \Delta P.A) \\
\vdash [P := \delta P.A]A & (\text{definition of } \Phi)
\end{array}$$

(ii). We have

$$\begin{array}{ll}
\vdash A \wedge \neg P(x) \to [P^+ := x^c]A & (\text{monotonicity of} \\
 & \quad \bigwedge Q.[P^+ := Q]A) \\
\vdash A \to \forall x\big(\neg[P^+ := x^c]A \to P(x)\big) & (\text{contraposition}) \\
\vdash A \to \{x \mid \neg[P^+ := x^c]A\} \subseteq P & (\text{definition of } \subseteq) \\
\vdash A \to (\Delta P.A)P \subseteq P & (\text{definition of } \Delta P.A) \\
\vdash A \to \delta P.A \subseteq P & (\text{by } (2))
\end{array}$$

□

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

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

$$\mathrm{Adm}(P) =_{\mathrm{def}} \{A \mid \neg[P^+ := x^c]A\epsilon Cts(P,x) \quad \text{and}$$

$$[P^- := Q]A\epsilon\ Pres(P)\}$$

where

$$\begin{array}{ll}
Cts(P,x) & =_{\mathrm{def}} \{A \mid \Lambda P.\{x \mid A\} \text{ is continuous}\}, \\
Pres(P) & =_{\mathrm{def}} \{A \mid \Lambda P.A \text{ is } \cap\text{-preserving}\}.
\end{array}$$

The following properties of Cts and $Pres$ are proved easily.

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

$$\bigvee \exists\big(A \wedge P(..) \wedge \ldots \wedge P(..)\big) \quad (P \text{ not in } A),$$

i.e. all (finite or infinite) disjunctions of existentially quantified formulae $A_1 \wedge \ldots \wedge A_n$, 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

$$\bigwedge \forall\big(A \wedge t \downarrow P(t)\big) \ (P \text{ not negatively in } A),$$

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

Now we define: a *Horn formula* in P is a formula of the form

$$\bigwedge \forall\big(A \wedge P(..) \wedge \ldots \wedge P(..)t \downarrow \rightarrow P(t)\big) \wedge \quad (P \text{ not in } A),$$

i.e. a (finite or infinite) conjunction of universally quantified formulae of the form $A_1 \wedge \ldots \wedge A_m \wedge t_1 \downarrow \wedge \ldots \wedge t_n \downarrow \rightarrow P(t_1, \ldots, t_n) \quad (m, n \geq 0)$, 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

$$\text{Horn}(P) \subseteq \text{Adm}(P).$$

In the sequel, we shall allow ourselves some sloppiness in the use of the phrase " $\cdots \epsilon$ 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

$$\bigwedge \forall\big(A \wedge P(..) \wedge \ldots \wedge P(..) \wedge t \downarrow \rightarrow P(s, t)\big),$$

although such formulae are in fact only $\equiv P$-equivalent to elements of Horn(P), hence in 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 $S_1 \times \ldots \times S_n \times S_{n+1}$, give an explicit definition of F and put $f(t_1, \ldots, t_n) := \iota x : S_{n+1}.F(t_1, \ldots, t_n, x)$. 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

$$\begin{array}{ll}
\epsilon(t) & =_{\text{def}} T \text{ if } f \text{ not in } t \\
\epsilon\big(f_i(t)\big) & =_{\text{def}} F(t^F, x_i) \\
\epsilon\big(g(t)\big) & =_{\text{def}} \epsilon(t)(= \epsilon(t_1) \wedge \ldots \wedge \epsilon(t_m)) \\
 & \quad \text{if } g \text{ different from } f
\end{array}$$

$$\begin{array}{ll}
t^F & =_{\text{def}} t \text{ if } f \text{ not in } t \\
\big(f_i(t)\big)^F & =_{\text{def}} x_i \\
\big(g(t)\big)^F & =_{\text{def}} g(t^F) \text{if } g \text{ different from } f
\end{array}$$

$$\begin{array}{l}
P(t)^F =_{\text{def}} P(t) \text{ if } f \text{ not in } t \\
P(t)^F =_{\text{def}} \exists x(\epsilon(t) \wedge P(t^F)) \text{ otherwise,}
\end{array}$$

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 $^\iota$ of descriptions mentioned in 3.2.3. It is straightforward that

$$(f \leftrightarrow F) \qquad \vdash \forall xy\big(f(x) = y \leftrightarrow F(x,y)\big) \to (A \leftrightarrow A^F)$$

holds. We also put

$$\begin{array}{ll} \delta f.A & =_{\text{def}} \lambda x.\iota y.(\delta F.A^F)(x,y). \\ \text{Adm}(f) & =_{\text{def}} \{A \mid A^F \epsilon \text{Adm}(F)\}. \\ A \equiv_f B & =_{\text{def}} A^F \equiv_F B^F. \end{array}$$

Theorem. Let $A\epsilon\text{Adm}(f)$.

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 $A, B\epsilon\text{Adm}(f)$ and $A \equiv B$, then $\delta f.A \equiv \delta f.B$.

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

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

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

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

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

$$\begin{array}{l} \vdash \forall xy\big(\iota y.(\delta F.A^F)(x,y) = y \leftrightarrow (\delta F.A^F)(x,y)\big) \\ \quad \to [f := \delta f.A]A; \end{array}$$

since

$$\begin{array}{l} \vdash \text{Func}(\delta F.A^F) \\ \quad \to \forall xy\big(\iota y.(\delta F.A^F)(x,y) = y \leftrightarrow (\delta F.A^F)(x,y)\big), \end{array}$$

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

$$\begin{aligned}&\vdash \mathrm{Func}(\delta F.A^F)\\ &\quad\to (\delta F.A^F \subseteq \{(x,y) \mid f(x)=y\} \leftrightarrow \lambda x.\iota y.(\delta F.A^F)(x,y) \subseteq f)\end{aligned}$$

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

iii) If $A, B \epsilon \mathrm{Adm}(f)$ then $A^F, B^F \epsilon \mathrm{Adm}(F)$, so $\delta F.A^F \equiv \delta F.B^F$ and this implies $\delta f.A \equiv \delta f.B$. □

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

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

$$(+) \qquad \forall x(x+0=x) \wedge \forall xy\big(x+Sy = S(x+y)\big)$$

Transforming this in the format used in this section yields

$$A =_{\mathrm{def}} \forall x\big(f(x,0)=x\big) \wedge \forall xy\Big(f(x,Sy) = S\big(f(x,y)\big)\Big), \text{ so}$$

$$\begin{aligned}A^F &\equiv_F \forall xF(x,0,x) \wedge \forall xy\exists uv(F(x,Sy,u) \wedge F(x,y,v) \wedge u = Sv)\\ &\equiv_F \forall xF(x,0,x) \wedge \forall xy\exists v\big(F(x,Sy,Sv) \wedge F(x,y,v)\big)\end{aligned}$$

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

$$\begin{aligned}\Delta F.A^F &= \Lambda F.\{(x,y,z) \mid (x=z \wedge y=0)\\ &\qquad \vee \exists w\forall v\big((y=Sw \wedge z=Sv) \vee (y=w \wedge z=v)\big)\}\\ &= \Lambda F.\{(x,0,x) \mid x=x\},\end{aligned}$$

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 $s \hookrightarrow t$ defined in 3.2.3 as an abbreviation of $\forall x(t=x \to s=x)$ (suggestively: if t is defined, then s is defined in the same way). The defining formula of addition can be written as

$$B =_{\mathrm{def}} \quad \forall x(f(x,0) \hookrightarrow x) \wedge \forall xy\Big(f(x,Sy) \hookrightarrow S\big(f(x,y)\big)\Big)$$

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

$$\begin{aligned}B^F &= \forall xF(x,0,x) \wedge \forall xyz(\exists u(F(x,y,u) \wedge Su=z)\\ &\qquad \to \exists v\big(F(x,Sy,v) \wedge v=z\big)\big)\\ &\qquad \equiv_F \forall xF(x,0,x) \wedge \forall xyu\big(F(x,y,u) \to F(x,Sy,Su)\big),\end{aligned}$$

so

$$\begin{aligned}\Delta F.B^F \quad &\equiv \Lambda F.\{(p,q,r) \mid \exists x(p = r = x \wedge q = 0)\\ &\quad \vee \exists xyu(F(x,y,u) \wedge p = x \wedge q = Sy \wedge r = Su)\}\\ &= \Lambda F.\{(x,0,x) \mid x = x\} \cup \{(x,Sy,Sz) \mid F(x,y,z)\}.\end{aligned}$$

To see that the fixpoint $\delta F.B^F$ of $\Delta F.B^F$ is functional, it suffices (using the fact that $\emptyset$ 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 $Sx = Sy \to x = y$ of S.

A *Horn formula* in f is a formula of the form

$$\begin{aligned}&\bigwedge \forall\big(A \wedge P_1(..f..) \wedge \ldots \wedge P_n(..f..) \wedge s \downarrow\!\to f(s) \hookrightarrow t(..f..)\big)\\ &\quad (f \text{ not in } A, s)\end{aligned}$$

i.e. a (finite or infinite) conjunction of universally quantified formulae of the form $A_1 \wedge \ldots \wedge A_m \wedge s_1 \downarrow \wedge \ldots \wedge s_n \downarrow\!\to f(s_1,\ldots,s_n) \hookrightarrow t \quad (m,n \leq 0)$, 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 $\mathrm{Horn}(f)$. We have

$$\mathrm{Horn}(f) \subseteq \mathrm{Adm}(f).$$

To see this, consider a typical element of $\mathrm{Horn}(f)$: $A =_{\mathrm{def}} \forall y(B \wedge P(f(t_1) \wedge t_2 \downarrow\!\to f(t_2) \hookrightarrow g(f(t_3)))$. Now $A^F = \forall y(B \wedge \exists x_1(F(t_1,x_1) \wedge P(x_1)) \wedge t_2 \downarrow\!\to \forall z(\exists x_3(F(t_3,x3) \wedge z = g(x3)) \to \exists x_2(F(t_2,x_2) \wedge z = x_2)))$, so

$$\begin{aligned}A^F \equiv_F \quad &\forall x_1x_3yz(B \wedge F(t_1,x_1) \wedge P(x_1) \wedge t_2 \downarrow\\ &\quad \wedge F(t_3,x_3) \wedge z = g(x_3) \to F(t_2,z));\end{aligned}$$

this last formula belongs to $\mathrm{Horn}(F)$, hence to $\mathrm{Adm}(F)$, so $A\epsilon\mathrm{Adm}(f)$.

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

$$A = \forall x\big(f(x) \hookrightarrow t(f,x)\big),$$

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

$$f(x) \simeq \iota y.(t(f,x) = y)$$

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: $\{\!\{.\}\!\}$ for the declarative and $[\![.]\!]$ 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:

`fbody` for function bodies,
`Pbody` for predicate bodies.

$$
\begin{array}{ll}
\{\!\{ \texttt{SORT S} \}\!\} & =_{\text{def}} \{\texttt{S}\} \\
\{\!\{ \texttt{FUNC f : S1 \# ... \# Sn -> S fbody} \}\!\} & =_{\text{def}} \{\texttt{f : S1} \times \ldots \times \texttt{Sn} \rightarrow \texttt{S}\} \\
\{\!\{ \texttt{PRED P : S1 \#... \# Sn Pbody} \}\!\} & =_{\text{def}} \{\texttt{P:S1} \times \ldots \times \texttt{Sn}\} \\
\{\!\{ \texttt{AXIOM A} \}\!\} & =_{\text{def}} \emptyset \\
[\![\texttt{SORT S}]\!] & =_{\text{def}} \top \\
[\![\texttt{AXIOM A}]\!] & =_{\text{def}} [\![\texttt{A}]\!]
\end{array}
$$

$$
\begin{array}{ll}
[\![\texttt{FUNC f: S1 \#... \# Sn} \rightarrow \texttt{PAR x1... xn DEF t}]\!] & \\
\quad =_{\text{def}} \forall x_1 : S_1 \ldots \forall x_n : S_n (f(x_1, \ldots, x_n) \simeq [\![\texttt{t}]\!]) & \\
[\![\texttt{FUNC f: S1 \#... \# Sn} \rightarrow \texttt{S IND A}]\!] & \\
\quad =_{\text{def}} \forall x_1 : S_1 \ldots \forall x_n : S_n \big(f(x_1, \ldots, x_n) & \\
\qquad \simeq (\delta f.[\![\texttt{A}]\!])(x_1, \ldots, x_n)\big) & \\
[\![\texttt{PRED P: S1 \#... \# Sn PAR x1... xn DEF A}]\!] & \\
\quad =_{\text{def}} \forall x_1 : S_1 \ldots \forall x_n : S_n (P(x_1, \ldots, x_n) \leftrightarrow [\![\texttt{A}]\!]) & \\
[\![\texttt{PRED P: S1 \#... \# Sn IND A}]\!] & \\
\quad =_{\text{def}} \forall x_1 : S_1 \ldots \forall x_n : S_n (P(x_1, \ldots, x_n) & \\
\qquad \leftrightarrow \big(\delta P.[\![\texttt{A}]\!])(x_1, \ldots, x_n)\big) &
\end{array}
$$

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 form 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 MPL_ω: 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 ρ 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> TO <sort-name>
        | <predicate-name> TO <predicate-name>
```

```
        | <function-name> TO <function-name>

<signature> ::= <item-list ','>
        | <renaming> @ <signature>
        | <signature> + <signature>
        | <item> ∧^signature>
        | SIG <scheme>

<item> ::= SORT <sort-name>
        | FUNC <typed function>
        | PRED <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 Σ. 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 Γ be a collection of sentences of L, Σ a signature, then the *closure of* Γ in Σ is defined by

$$Cl(\Sigma, \Gamma) =_{\text{def}} \{A \mid \text{MPL}_\omega : \Gamma \vdash A \text{ and } \text{SIG}(A) \subseteq c(\Sigma)\}$$

These closures are called *theories*. TH is defined as the collection of all theories of MPL_ω.

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:

$$\Sigma \,\square\, T =_{\text{def}} Cl(\Sigma \cap SIG(T), T).$$

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

$$
\begin{array}{ll}
dom(\rho) & =_{\text{def}} \{I \mid \rho(I) \neq I\} \\
rg(\rho) & =_{\text{def}} \{\rho(I) \mid \rho(I) \neq I\}
\end{array}
$$

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

$$
\begin{array}{ll}
inj(\rho, \Sigma, \Pi) & =_{\text{def}} \forall I \epsilon \Sigma \forall J \epsilon \Pi (\rho(I) = \rho(J) \rightarrow I = J) \\
inj(\rho, \Sigma) & =_{\text{def}} inj(\rho, \Sigma, \Sigma) \\
inj(\rho) & =_{\text{def}} inj\big(\rho, dom(\rho)\big)
\end{array}
$$

We also put:

ρ *renames* Σ *outside* Π iff
$dom(\rho) = \Sigma \cap \Pi, rg(\rho) \cap (\Sigma \cup \Pi) = \emptyset$ and $inj(\rho, \Sigma)$.

We shall use the *good renaming property*:

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

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

The application of renamings on theories is defined by

$$\rho(T) =_{\text{def}} Cl\big(\rho(S(T)), \{\rho(A) \mid A \epsilon T\}\big).$$

It is clear that $\rho(T) = \{\rho(A) \mid A \epsilon T\}$ if $inj(\rho, SIG(T))$.

The operators introduced above satisfy several properties, of which the least trivial read as follows. Let T, U, V range over theories, Σ and Π over signatures, ρ and σ over renamings.

$$
\begin{array}{ll}
(\ \Box\ \Box\ ) & \Sigma \ \Box\ (\Pi \ \Box\ T) = (\Sigma \cap \Pi) \ \Box\ T \\
(\rho\ \Box\ ) & inj(\rho, SIG(T) - \Sigma, SIG(T) \cup \Sigma) \Rightarrow \rho(\Sigma \ \Box\ T) = \rho(\Sigma) \ \Box\ \rho(T) \\
(\ \Box\ \rho) & \big(dom(\rho) \cup rg(\rho)\big) \cap \Sigma = \emptyset \quad \& \quad inj\big(\rho, SIG(T)\big) \Rightarrow \Sigma \ \Box\ T \\
 & \qquad = \Sigma \ \Box\ \rho(T) \\
(\ \Box +) & SIG(T) \cap SIG(U) \subseteq \Sigma \Rightarrow \Sigma \ \Box\ (T + U) = \Sigma \ \Box\ T + \Sigma \ \Box\ U \\
(\gamma) & dom\big(\gamma(\Sigma, \Pi)\big) = \Sigma \cap \Pi \quad \& \quad rg\big(\gamma(\Sigma, \Pi)\big) \cap (\Sigma \cup \Pi) = \emptyset \\
 & \quad \& \ inj(\gamma(\Sigma, \Pi), \Sigma)
\end{array}
$$

The condition in $(\rho\ \Box\ )$ is required to prevent new identifications of names in $\rho(T)$ and between names in $\rho(\Sigma)$ and $\rho(T)$ not present in $\rho(\Sigma \ \Box\ T)$; the condition in $(\ \Box\ \rho)$ guarantees that ρ does not affect names in $\Sigma \ \Box\ T$, neither introduces new identifications in T.

Most of the properties listed have a straightforward proof; only the proof of $(\ \Box +)$ is more involved and makes essential use of the Interpolation Property of MPL_ω.

Now define the language TL of *theory terms* by

$$T ::= Th \mid \rho(T) \mid \Sigma \ \Box\ T \mid T + T,$$

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

$$\Sigma \ \square \ \big(\rho_1(Th_1) + \ldots + \rho_n(Th_n)\big).$$

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

1) for any Σ, T, ρ there is a σ with $\rho(\Sigma \ \square \ T) = \rho(\Sigma) \ \square \ \sigma(T)$;
2) for any Σ, Π, T, U, there are ρ, σ with $\Sigma \ \square \ T + \Pi \ \square \ U = (\Sigma \cup \Pi) \ \square \ \big(\rho(T) + \sigma(U)\big)$.

(1) relies on (γ), ($\square$ ρ) and (ρ $\square$); (2) on ($\square$ ρ) and ($\square$ $+$). 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 $T \epsilon TH$, then $a[X \to T]$ is the assignment wich 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:

`k l`	schemes
`X`	scheme variables
`D`	definitions
`R`	renamings
Σ	signatures
`S`	sort names
`I J`	items

$$
\begin{array}{ll}
[\![\texttt{X}]\!](a) & =_{\text{def}} a(X) \\
[\![\ \texttt{CLASS D}_1 \ldots \texttt{Dn END}\ ]\!](a) & =_{\text{def}} Cl\big((\{\!\{\texttt{D}_1\}\!\} \cup \ldots \cup \{\!\{\texttt{Dn}\ \}\!\}), \\
 & \qquad \{[\![\texttt{D}_1]\!], \ldots, [\![\texttt{Dn}\ ]\!]\}\big) \\
[\![\ \texttt{RENAME R IN k}]\!](a) & =_{\text{def}} [\![\texttt{R}]\!]\big([\![\texttt{k}]\!](a)\big) \\
[\![\ \texttt{IMPORT k INTO l}]\!](a) & =_{\text{def}} [\![\texttt{k}]\!](a) + [\![\texttt{l}]\!](a) \\
[\![\ \texttt{EXPORT } \Sigma \texttt{ FROM k}\ ]\!](a) & =_{\text{def}} [\![\Sigma]\!](a) \ \square \ [\![\texttt{k}\ ]\!](a) \\
[\![\ \texttt{LET X := k ; l}\ ]\!](a) & =_{\text{def}} [\![\texttt{l}\ ]\!](a[X \to [\![k]\!](a)]) \\
[\![\texttt{I1 TO J1}, \ldots, \texttt{In TO Jn}]\!] & =_{\text{def}} [[\![\texttt{I1}]\!] := [\![\texttt{J1}]\!], \ldots, \\
 & \qquad [\![\texttt{In}]\!] := [\![\texttt{Jn}]\!]] \\
[\![\texttt{R \$ S}]\!] & =_{\text{def}} [\![\texttt{R}]\!] \circ [\![\texttt{S}]\!] \\
[\![\texttt{I1}, \ldots, \texttt{In}]\!](a) & =_{\text{def}} \{[\![\texttt{I1}]\!](a), \ldots, [\![\texttt{In}]\!](a)\} \\
[\![\texttt{R @}\Sigma]\!](a) & =_{\text{def}} [\![\texttt{R}]\!]\big([\![\Sigma]\!](a)\big) \\
[\![\Sigma_1 + \Sigma_2]\!](a) & =_{\text{def}} [\![\Sigma_1]\!](a) \cup [\![\Sigma_2]\!](a) \\
[\![\texttt{I ^}\Sigma]\!](a) & =_{\text{def}} [\![\Sigma]\!](a) - \{[\![\texttt{I}\ ]\!](a)\} \\
[\![\ \texttt{SIG k}]\!](a) & =_{\text{def}} SIG\big([\![\texttt{k}\ ]\!](a)\big)
\end{array}
$$

$$
\begin{array}{ll}
[\![\texttt{SORT S}]\!](a) & =_{\text{def}} S \\
[\![\texttt{FUNC f : S1 \# ... \# Sn -> S}]\!](a) & =_{\text{def}} f \\
[\![\texttt{PRED P : S1 \# ... \# Sn}]\!](a) & =_{\text{def}} P
\end{array}
$$

3.5 The parametrized scheme language

3.5.1 Introduction

Parametrisation is obtained in 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 *implements* A. For flat schemes C and A, we have that C implements A iff the meaning of C (a theory in MPL_ω) 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

```
<parscheme> ::= <parscheme-var>
      | <scheme>
      | LAMBDA <parscheme-var> : <parscheme> OF <parscheme>
      | APPLY <parscheme> TO <parscheme>
      | LET <parscheme-var> := <parscheme> ; <parscheme>
```

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 $TH^\omega = \{TH^\tau \mid \tau\epsilon \text{ type}\}$ 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^σ to TH^τ.

For $\tau\epsilon$type, the ordering $\leq^\tau$ on TH^τ with least element $\perp^\tau$ is defined by

$$
\begin{array}{ll}
\leq^0 & =_{\text{def}} \subseteq \\
\Phi \leq^{\sigma\rightarrow\tau} \Psi & \text{iff } \forall T\epsilon TH^\sigma \big(\Phi(T) \leq^\tau \Psi(T)\big) \\
\perp^0 & =_{\text{def}} Cl(\emptyset, \emptyset) \\
\perp^{\sigma\rightarrow\tau} & =_{\text{def}} \lambda T\epsilon TH^\sigma .\ \perp^\tau
\end{array}
$$

We extend the theory language TL of 3.4.3 to TTL, *typed theory language*, containing terms T^τ of type $\tau\epsilon$type:

$$
\begin{array}{ll}
T^0 :: & = Th \mid X^0 \mid \rho(T^0) \mid \Sigma \ \Box T^0 \mid T^0 + T^0 \\
T^t :: & = X^\tau \mid T^{\sigma\to\tau} T^\sigma \\
T^{\sigma\to\tau} :: & = \lambda X^\sigma \geq^\sigma T^\sigma . T^\tau
\end{array}
$$

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

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 $\bot$.

Substitution $[X := A]B$ is defined with α-conversion, so

$$
\begin{array}{ll}
[X := A](\lambda Y \geq B.C) & =_{\text{def}} \lambda Y \geq [X := A]B.[X := A]C \\
& \quad \text{if } Y \text{ not free in } A \\
& =_{\text{def}} \lambda Z \geq [X := A][Y := Z]B.[X := A] \\
& \quad [Y := Z]C \text{ if } Y \text{ free in } A,
\end{array}
$$

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

Besides the properties listed above, we have here:

$$
\begin{array}{ll}
(\leq^\tau po) & \leq^\tau \text{ is a partial order with least element } \bot^\tau \\
(\leq ap) & A \leq B \to AC \leq BC \\
(\leq \lambda 1) & A \leq B \to (\lambda X \geq A.C) \geq (\lambda X \geq B.C) \\
(\leq \lambda 2) & \forall X (X \geq A \to B \leq C) \to (\lambda X \geq A.B) \leq (\lambda X \geq A.C) \\
(\pi) & (C \geq A \to (\lambda X \geq A.B)C = [X := C]B) \wedge \\
& \quad (C \geq A \vee (\lambda X \geq A.B)C = \bot) \\
(\eta) & \forall X (X \geq A \vee BX = \bot) \to \lambda X \geq A.BX = B \\
& \quad (\text{provided } X \text{ not free in } B)
\end{array}
$$

Observe that not all terms of higher type denote monotonic functions: $\lambda X.\lambda Y \geq X.B$ is in general not monotonic, and antimonotonic if X does not occur in B. (π) is the partial version of the β-conversion axiom $(\lambda X.A)B = [X := B]A$ of the λ-calculus, (η) is a variant of the η-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 λ-calculus with abstraction terms $\lambda X \geq A.B$; the axiomatics is given in the form of a derivation system with sequents $\Gamma \vdash A \leq B$, where Γ 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)

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

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:

$(\Diamond)$ if $\Gamma \vdash A \to^* B, A \to^* C$, then $\Gamma \vdash B \to^* D, C \to^* D$ for some D.

The proof given in [11, 4.2.5.7] follows the lines of P. Martin-Löf's proof of the corresponding property for the λ-calculus, as given in [2, 3.2].

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) no term of applied $\lambda\pi$-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 λ-typed λ-calculus): define an embedding of the applied $\lambda\pi$-calculus into the typed λ-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 λ-calculus, and (SN) follows from the strong normalisation property of typed $\bigwedge$-calculus.

Reduction in TTL. Let A, B, C be TTL-terms of appropriate type. We call the transitions

$$\begin{array}{ll} (\lambda X \geq A.B)C \rightarrow & [X := C]B \\ (\lambda X \geq A.B)C \rightarrow & \bot \\ \lambda X \geq A.BX \rightarrow & B \end{array}$$

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 $\bot$), then the second reduction is valid; the third reduction is valid if $\forall X(X \geq A \vee BX = \bot)$ 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 λ-calculus; but that theory is strongly normalizing, so we have contradiction and conclude that (SN) holds for TTL.

By Newman's lemma (see [2, 3.1.25]), in the presence of (SN) the diamond property is equivalent to the weak diamond property:

$(W\Diamond)$ if $A \rightarrow A_1$ and $A \rightarrow A_2$, then $A_1 \rightarrow^* B$ and $A_2 \rightarrow^* B$ 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):

$$
\begin{array}{lll}
(1) & A \equiv BC, A' & \equiv B'C, B \to B' \\
(2) & A \equiv BC, A' & \equiv BC', C \to C' \\
(3) & A \equiv \lambda X \geq B.C, A' & \equiv \lambda X \geq B'.C, B \to B' \\
(4) & A \equiv \lambda X \geq B.C, A' & \equiv \lambda X \geq B.C', C \to C' \\
(5) & A \equiv (\lambda X \geq B.C)D, A' & \equiv [X := D]C, D \geq B \\
(6) & A \equiv (\lambda X \geq B.C)D, A' & \equiv \bot, D \not\geq B \\
(7) & A \equiv \lambda X \geq B.CX, A' & \equiv C, \forall X(X \geq B \vee CX = \bot)
\end{array}
$$

We consider a typical case:

(5,2): $A \equiv (\lambda X \geq B.C)D, A_1 \equiv [X := D]C, D \geq B$, $A_2 \equiv (\lambda X \geq B.C)D', D \to D'$. Take $A_3 := [X := D']C$, then $A_1 \to^* A_3$ and $A_2 \to^* A_3$, 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 $N \to N'$. As a direct consequence of (SN) and $(\Diamond)$, we have:

every term in TTL reduces to a unique normal form.

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

$$\lambda X.\lambda Y.\big((\lambda Z \geq X.A)Y\big);$$

this is a closed term containing the redex $(\lambda Z \geq X.A)Y$ 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^ω of corresponding type; if a is such an assignment and $T \epsilon TH^\omega$ has the same type as X, then $a[X \to T]$ is the assignment wich sends X to T and behaves like a on the other scheme variables. We use the following variables for COLD-K^2-constructs:

`K L M`	parschemes
`Y`	parscheme variables

We define

$$
\begin{array}{ll}
[\![\texttt{Y}]\!](a) & =_{\text{def}}\ a(Y) \\
[\![\texttt{LAMBDA Y : K OF L}]\!](a) & =_{\text{def}}\ \lambda T \geq [\![\texttt{K}]\!](a).[\![\texttt{L}]\!](a[\![\texttt{Y}\to\texttt{T}]\!]) \\
[\![\texttt{APPLY K TO L}]\!](a) & =_{\text{def}}\ [\![\texttt{K}]\!](a)[\![\texttt{L}]\!](a) \\
[\![\ \texttt{LET Y := K ; L}]\!](a) & =_{\text{def}}\ [\![\texttt{L}\ ]\!](a[\![\texttt{Y}\to\ [\![\texttt{K}]\!](a)]\!])
\end{array}
$$

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 a 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.

$$
\begin{aligned}
&[\![\ \texttt{DESIGN C1;}\ldots\texttt{;Cm SYSTEM M1,}\ldots,\texttt{Mn}]\!](a) =_{\text{def}} \\
&\qquad < [\![\ \texttt{DESIGN C1;}\ldots\texttt{; Cm SYSTEM M1}]\!](a), \\
&\qquad\quad \ldots, \\
&\qquad\quad [\![\texttt{DESIGN C1;}\ldots\texttt{; Cm SYSTEM Mn}]\!](a) >
\end{aligned}
$$

$$
[\![\ \texttt{DESIGN SYSTEM M}]\!](a) =_{\text{def}} [\![\texttt{M}]\!](a)
$$

$$
\begin{aligned}
&[\![\ \texttt{DESIGN Y : K := L; C1 ;}\ldots\texttt{; Cm SYSTEM M}]\!](a) =_{\text{def}} \\
&\qquad \big(\lambda T \geq [\![\texttt{K}]\!](a).[\![\texttt{DESIGN C1 ;}\ldots\texttt{; Cm SYSTEM M}]\!] \\
&\quad (a[Y \to T])\big)\big([\![\texttt{L}]\!](a)\big)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ \texttt{DESIGN Y : K; C1 ;}\ldots\texttt{;Cm SYSTEM M}]\!] =_{\text{def}} \\
&\quad \Lambda T \geq [\![\texttt{K}]\!](a).[\![\texttt{DESIGN C1 ;}\ldots\texttt{;Cm SYSTEM M}]\!](a[Y \to T])
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ \texttt{DESIGN LET Y : K; C1 ;}\ldots\texttt{; Cm SYSTEM M}]\!] =_{\text{def}} \\
&\quad [\![\texttt{DESIGN C1 ;}\ldots\texttt{; Cm SYSTEM M}]\!](a[Y \to [\![\texttt{K}]\!](a)])
\end{aligned}
$$

References

[1] J.C.M. Baeten, J.A. Bergstra, S. Mauw, G.J. Veltink, *A process specification formalism based on static* COLD, in: J.A. Bergstra, L.M.G. Feijs (eds.) *Algebraic methods: Theory, Tools and Applications* (Part II), LNCS, Springer-Verlag, 1991

[2] H.P. Barendregt, *The lambda calculus, its syntax and semantics* (Revised edition), North-Holland, 1984

[3] H. Barringer, J.H. Cheng, C.B. Jones, A logic covering undefinedness in program proofs, Acta Informatica 21 (1984) 251 - 269.

[4] J.A. Bergstra, J. Heering, P. Klint, *Module algebra*, Journal of the ACM 37 (1990) 335 - 372

[5] J.H. Cheng, A Logic for Partial Functions, Ph.D. thesis, University of Manchester, Department of Computer Science, 1986 Technical Report UMCS-86-7-1.

[6] M.E.A. Corthout, H.B.M. Jonkers, *The transformational development of a new point containment algorithm*, Philips Journal of Research 41 (1986), 83-174

[7] M.E.A. Corthout, H.B.M. Jonkers, *A new point containment algorithm for B-regions in the discrete plane*, Proceedings of the Conference on Theoretical Foundations of Computer Graphics and CAD, Italy (1987)

[8] L.M.G. Feijs, A formalisation of design methods, Ph.D. thesis, Technical University Eindhoven, 1990.

[9] L.M.G. Feijs, H.B.M. Jonkers, *Transformational design: an annotated example*, Proceedings of the IFIP TC 2 Working Conference on Program Specification and Transformation, North-Holland (1986)

[8] L.M.G. Feijs, A formalisation of design methods, Ph.D. thesis, Technical University Eindhoven, 1990.

[10] L.M.G. Feijs, H.B.M. Jonkers, *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science 35, Cambridge University Press, 1992.

[11] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans, G.R. Renardel de Lavalette, *Formal definition of the design language* COLD-K, Technical Report, ESPRIT project 432, Doc.Nr. METEOR/t7/PRLE/7 (1987)

[12] R. Groenboom, G.R. Renardel de Lavalette, *Reasoning over dynamic features in specification languages*, this volume.

[13] D. Harel, *Dynamic Logic*, in: Handbook of Philosophical Logic, vol. II (D. Gabbay, F. Guenthner, eds.) Reidel (1984) 497–604.

[14] H.B.M. Jonkers, *Inheritance in* COLD, in: J.A. Bergstra, L.M.G. Feijs (eds.) Algebraic methods: Theory, Tools and Applications (Part II), LNCS, Springer-Verlag, 1991

[15] H.B.M. Jonkers, J.H. Obbink, COLD: *a common object-oriented language for design*, Working document, Philips Research Laboratories Eindhoven, 1983

[16] C.P.J. Koymans, G.R. Renardel de Lavalette, *The logic* MPL_ω, in: Algebraic methods: Theory, Tools and Applications (M. Wirsing, J.A. Bergstra, eds.) Lecture Notes in Computer Science 394, Springer-Verlag, 1989

[17] C.A. Middelburg, *The* VIP VDM *specification language*, in: VDM'88, VDM - the way ahead (R. Bloomfield, L. Marshall, R. Jones, eds.), Lecture Notes in Computer Science 328, Springer-Verlag (1989) 187 - 201

[18] C.A. Middelburg, VVSL: *a language for structured* VDM *specifications*, Formal aspects of Computing 1 (1989) 115 - 135

[19] C.A. Middelburg, G.R. Renardel de Lavalette, LPF *and* MPL_ω *- a logical comparison of* VDM SL *and* COLD-K, in: S. Prehn, W.J. Toetenel

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

[20] G.R. Renardel de Lavalette, COLD-K^2, *the static kernel of* COLD-K, Report RP/mod-89/8, Software Engineering Research Centre, Utrecht, the Netherlands, November 1989

[21] G.R. Renardel de Lavalette, *Logical semantics of modularisation*, in: Computer Science Logic '91 (E. Börger, G. Jäger, H. Kleine Büning, M.M. Richter, eds.), LNCS626, 306 - 315

[22] G.R. Renardel de Lavalette, *Formal development of a serial copy management system*, in: VDM'91: Formal Software Development Methods (S. Prehn, W.J. Toetenel, eds.), LNCS551, Springer-Verlag, 477 - 495

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

[24] D.S Scott, *Existance and description in Formal Logic*, in: R. Schoenman (Ed.), Bertrand Russell, Philosopher of the Century, Allen & Unwin, London, (1967), 181–200.

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 `LET x1:=t1, ... , xn:=tn`, the object variables `x1`, ... `xn` are mutually different.

ii) In the definitions

```
FUNC f:  S1 # ... # Sm → S PAR x1 ... xn  DEF t
FUNC f:  S1 #... # Sm → S PAR x1... xn  IND A1
PRED P: S1 #... # Sm PAR x1 ... xn  DEF A2
PRED P: S1 # ... # Sm  PAR x1... xn  IND A3
AXIOM A4
```

we have:

- m equals n and the variables $x_1 \ldots x_n$ are all different;
- the free variables in t, A_1, A_2, A_3 are among $x_1 \ldots x_n$;
- A_4 is closed, i.e. does not contain free variables;
- f does not occur in t, and P not in A_2.

iii) For every function symbol f occurring in CLASS $D_1 \ldots D_n$ END, there is exactly one i $(1 \leq i \leq n)$ such that D_i equals FUNC $f : S_1 \# \ldots \# S_m \rightarrow S$ $f$body, where $f$body may be empty; this D_i is called the definition of f; 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 TO $J_1, \ldots, I_n$ TO J_n

are well-formed if:

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

vi) In LAMBDA $Y : K$ OF L the scheme variable Y 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 MPL - (ι).

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

$$
\begin{array}{ll}
(\uparrow_S)^L & =_{\text{def}} \uparrow \\
t^L & =_{\text{def}} t \text{ for all other terms } t \\
(s =_S t)^L & =_{\text{def}} (S(s) \wedge s = t) \\
(t \downarrow_S)^L & =_{\text{def}} S(t) \\
\big(P(t)\big)^L & =_{\text{def}} P(t^L) \text{ for all other predicates } P \\
(\neg A)^L & =_{\text{def}} \neg A^L \\
(A \wedge B)^L & =_{\text{def}} A^L \wedge B^L \\
\big(\forall x : S(A)\big)^L & =_{\text{def}} \forall x(S(x) \to A^L)
\end{array}
$$

Abbreviating $S_1(x_1) \wedge \ldots \wedge S_n(x_n)$ by $\mathbf{S}(\mathbf{x})$, we also put

$$
\begin{array}{lll}
Ax(S) & =_{\text{def}} \neg S(\uparrow) & \\
Ax(f) & =_{\text{def}} \forall \mathbf{x}\big(S\big(f(\mathbf{x})\big) \to \mathbf{S}(\mathbf{x})\big) & \\
 & \text{if type}(f) \quad = S_1 \times \ldots \times S_n \to S & \\
Ax(P) & =_{\text{def}} \forall \mathbf{x}\big(P(\mathbf{x}) \to \mathbf{S}(\mathbf{x})\big) & \\
 & \text{if type}(P) \quad = S_1 \times \ldots \times S_n & \\
Ax(\Sigma) & =_{\text{def}} \{Ax(I) \mid I \epsilon \Sigma\} & \text{for MPL-signatures } \Sigma.
\end{array}
$$

Now we have

$$\text{MPL}(\Sigma) - (\iota) \vdash A \Leftrightarrow L(\Sigma \cup \{\uparrow\}) : Ax(\Sigma) \vdash A^L$$

$\Rightarrow$ 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 $M' \models A_x(\Sigma)$ 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 $\Sigma = \{S, S', f : S \to S', P : S\}$, then $M = \{S^M, S'^M, f^M, P^M\}$; now $M_A = \{D, \underline{f}, \underline{S}, \underline{S'}, \underline{P}, \underline{*}\}$ is given by

$$
\begin{array}{ll}
D & =_{\text{def}} \underline{S} \cup \underline{S'} \cup \{\underline{*}\} \\
\underline{f} & =_{\text{def}} f^M \cup \{< d, \underline{*}\} > \mid d \epsilon \underline{S'} \cup \{\underline{*}\}\} \\
\underline{S} & =_{\text{def}} S^M - \{^* S^M\} \\
\underline{S'} & =_{\text{def}} S'^M - \{^* S'^M\} \\
\underline{P} & =_{\text{def}} P^M.
\end{array}
$$