Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering, P. Klint

# Towards shorter algebraic specifications: a simple language definition and its compilation to Prolog

Computer Science/Department of Software Technology Report CS-R8814 April

69D21, 69D22, 69D41, 69D43, 69F32

# Towards Shorter Algebraic Specifications: a Simple Language Definition and its Compilation to Prolog

J. Heering
*Department of Software Technology, Centre for Mathematics and Computer Science*
*P.O. Box 4079, 1009 AB Amsterdam, The Netherlands*

P. Klint
*Department of Software Technology, Centre for Mathematics and Computer Science*
*P.O. Box 4079, 1009 AB Amsterdam, The Netherlands*
*and*
*Programming Research Group, University of Amsterdam*
*P.O. Box 41882, 1009 DB Amsterdam, The Netherlands*

Algebraic specifications tend to become obscured by lengthy definitions of relatively trivial cases. This occurs, for instance, in specifications of predicates on finite domains and in specifications dealing with error handling. In this paper we introduce several abbreviation techniques and apply them in a new specification of the toy language PICO. This specification has four major characteristics: (1) it is written in a combination of ASF and SDF (the former is used for defining all semantic aspects of PICO, the latter for defining all syntactic aspects); (2) no error cases are specified; (3) both positive and negative conditions are allowed in equations; (4) it can be translated to Prolog using an extension of the compilational approach of van Emden and Yukawa.

*1987 CR Categories:* D.2.1 **[Software Engineering]**: Requirements/Specifications - *Languages;* D.2.2 **[Software Engineering]**: Tools and Techniques - *Modules and Interfaces;* D.3.1 **[Programming Languages]**: Formal Definitions and Theory - *Syntax, Semantics;* D.3.3 **[Programming Languages]**: Language Constructs - *Modules;* F.3.2 **[Logics and Meanings of Programs]**: Semantics of Programming Languages - *Algebraic Approaches to Semantics.*

*1985 Mathematics Subject Classification:* 68N15 **[Software]**: Programming languages; 68Q55 **[Theory of computation]**: Semantics.

*Key Words & Phrases:* algebraic definition of programming languages, user-defined syntax, typechecking and evaluation of programs, inequality, negative condition, quasi-initial model, partial function, partial algebra, error handling, compilation of algebraic specifications to Prolog.

*Note:* Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

*Note:* This report is a minor revision of Chapter 9 of *Exercises in Algebraic Specification, eds. J.A. Bergstra, J. Heering & P. Klint, Addison-Wesley, to appear.*

## 1 INTRODUCTION

Two of the major shortcomings of the algebraic specification formalism ASF [BHK85, BHK87] are:

- the limited notational mechanisms for writing functions (only prefix notation and a very limited form of infix notation are allowed);
- the need to specify many, obviously uninteresting, cases.

The first problem has been amply discussed in earlier papers, and a solution to it was found in the form of the syntax definition formalism SDF introduced in [HK87]. An example of a specification using a combination of ASF and SDF was given in [Hen87]. In this paper we concentrate on the second problem: how can the specification of many uninteresting and trivial cases be avoided?

Report CS-R8814
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Let us introduce the subject by means of some examples. As a first example, consider the definition of an equality function on a sort `S` with a finite number of $n$ elements. A complete equational specification of the equality function `eq: S # S -> BOOL` requires $\Omega(n^2)$ equations (i.e., a number of equations quadratic in $n$): the single equation `eq(x,x) = true` specifying the *true* cases and all others specifying the *false* cases. As a second example, consider the definition of a typechecking function for a binary operator in a programming language with $n$ primitive types. An equational specification of this function will also require $\Omega(n^2)$ equations, of which only a few define *true* cases. Similar problems occur in specifications dealing with *error* cases: ordinarily, the number of error cases dominates the number of correct cases to be specified. In all these examples, the question arises how the corresponding specifications can be abbreviated.

Inspired by the conciseness of specifications written in the semantics definition formalism Typol [Des84, Kah87] (which was in turn inspired by Plotkin's *structural operational semantics* [Plo81]), we attempt to obtain specifications of a comparable size in the algebraic case. To this end we have taken the following steps: (1) we experiment with *negative conditions* in equations; (2) we completely omit the specification of error cases.

As to the first point, negative conditions are very powerful, allowing superlinear abbreviation of specifications in many cases, but they are somewhat troublesome from a semantic point of view in the sense that initial algebra semantics may become ambiguous if negative conditions are used. That is, the initial model, which was unique (up to isomorphism) in the positive conditional case, need no longer exist. Kaplan has shown that its place is taken by a collection of *quasi-initial* models which is never empty and generally quite large [Kap87b]. We borrow a simple example from Kaplan's paper to illustrate what is going on. Consider

```
module Simple1
begin
  exports
  begin
    sorts A
    functions
      a :         -> A
      b :         -> A
      c :         -> A
  end

  equations

  [1]   a = c when a ≠ b

end Simple1
```

where the single equation has negative condition `a ≠ b`. `Simple1` has the following minimal models (i.e., models without "junk"):

1. A two element model $M_1$ with `a ≠ b` and `a = c`.
2. A two element model $M_2$ with `a = b` and `a ≠ c`.
3. A one element model $M_3$ with `a = b = c`.

Of these three models, both $M_1$ and $M_2$ satisfy the "no confusion" criterion. They are both quasi-initial and there is no uniquely determined initial model. The, perhaps somewhat surprising, symmetry between models $M_1$ and $M_2$ is a reflection of the fact that

```
a = c when a ≠ b
```

is logically equivalent to

```
a = c ∨ a = b.
```

Kaplan argues that $M_1$ is the preferred model in this case and this is borne out by our own experience as will become clear in the sequel. Such a preferred quasi-initial model does not always exist, however. For instance,

```
module Simple2
begin
  exports
  begin
    sorts A
    functions
      a :        -> A
      b :        -> A
      c :        -> A
  end

  equations

  [1]  a = c when a ≠ b

  [2]  a = b when a ≠ c

end Simple2
```

has the same quasi-initial models as `Simple1`, but there is no longer reason to prefer $M_1$ over $M_2$ (or $M_2$ over $M_1$, for that matter).

In the case of `Simple1`, the equation `a = c` holds in the preferred quasi-initial model $M_1$, but it is not provable from `[1]` by means of the standard rules of deduction. This incompleteness (which has no parallel in the positive conditional case) can be repaired by using an additional rule of deduction. In our case, we use Prolog's *negation as failure* rule (in the form of the \== operator) [L84] to obtain a complete implementation of the preferred quasi-initial algebra semantics of our specification (see Section 3). See Kaplan's paper for further properties of quasi-initial models and the associated positive/negative conditional term rewriting systems.

Omitting error handling is the second means by which we hope to achieve shorter specifications. Unfortunately, this is not as simple as it may appear at first glance. All functions in a (quasi-)initial algebra are *total*. Defining a function only partially, does not lead to a partial function in the initial model, but to a total function and an (often very large) number of so-called *non-standard values*. The existence of these values is forgotten all too easily, leading to specifications that contain subtle errors. One intuitively appealing way out of this predicament is to switch to *partial algebra semantics* [Rei87] (see Sections 2.2.3 and 5). For another approach to error handling based on *subsorts*, and a survey of the various approaches that have been tried see [GJM85, GM85].

In this paper we also experiment with a different notation for conditional equations. Instead of writing

$$u = v \textbf{ when } u_1 = v_1 , \ldots , u_n = v_n$$

as we did earlier in ASF, we will now use a notation akin to the one in use for inference rules and write

$$\frac{u_1 = v_1 , \ldots , u_n = v_n}{u = v}.$$

An experiment with the new style of specification is presented in Section 2, where we define, once again, the static and dynamic semantics of PICO. The new specification can be compared with the original one given in [BHK85] provided that the following is taken into account. The original specification does not only define typechecking and evaluation based on an abstract syntax representation of PICO programs, but it also gives a complete definition of the process of parsing and constructing abstract syntax trees. The former may be compared with the specification given in this paper, the latter need not be specified here due to the use of SDF.

In Section 3 we give an implementation scheme for specifications in our new style, and in Section 4 we apply this scheme to the new PICO definition. Section 5 gives some conclusions and directions for further research.

## 2 A NEW SPECIFICATION OF PICO

### 2.1 Elementary data types

We will not give the complete specifications for the Booleans, natural numbers and strings here, but only list the required sorts and functions. Module Booleans defines sort BOOL with constants *true* and *false*. Module Naturals defines sorts NAT-CON (natural number constants) and NAT (arbitrary natural numbers), and operations + (addition) and – (cut-off subtraction). This specification is similar to the one given in [Hen87], except that we use full SDF in this paper and make a distinction between lexical and context-free syntax. Module Strings defines sorts STR-CON (string constants, the empty string is denoted by "") and STRING (arbitrary strings), and the operation | | (concatenation).

### 2.2 Identifiers, types, and tables

Next, we give specifications for PICO identifiers (Section 2.2.1), the type language of PICO (Section 2.2.2), and the table data type (Section 2.2.3).

#### 2.2.1 Identifiers

PICO identifiers consist of a letter followed by zero or more letters or digits. The specification is as follows:

```
module Identifiers
begin
   exports
   begin
      lexical syntax
         sorts   LETTER, LETTER-OR-DIGIT, ID
         functions
            [a-z]                          → LETTER
            [a-z0-9]                       → LETTER-OR-DIGIT
            LETTER LETTER-OR-DIGIT*        → ID
   end
end Identifiers
```

#### 2.2.2 Types

The type system of PICO is trivial. The only possible types are natural number or string. We define a predicate *compatible* on types, which is *true* when its arguments are either both *natural* or both *string*. All other possible cases are left undefined. Hence, *compatible* introduces non-standard Boolean values *compatible*(*natural*,*string*), etc. The specification is:

```
module Pico-types
begin
   exports
   begin
      context-free syntax
         sorts   TYPE
         functions
            natural                              → TYPE
            string                               → TYPE
            compatible "(" TYPE "," TYPE ")"     → BOOL
   end
   imports
      Booleans

   equations
      [Typ1] compatible(natural,natural) = true

      [Typ2] compatible(string,string) = true

end Pico-types
```

**2.2.3 Tables**

The table data type is a prerequisite for the definition of the type environment and value environment used during typechecking and evaluation. The specification of Tables is:

```
module Tables
begin
   parameters
      Entries
      begin
         sorts   ENTRY
      end Entries

   exports
   begin
      context-free syntax
         sorts   PAIR, TABLE
         functions
            ID ":" ENTRY        → PAIR {par}
            empty-table         → TABLE
            PAIR "⊕" TABLE      → TABLE
            lookup ID in TABLE  → ENTRY
   end

   imports
      Identifiers

   variables
      Id, Id′  → ID
      E        → ENTRY
      T        → TABLE

   equations
      [T1a] lookup Id in (Id:E) ⊕ T = E
```

$$[\text{T1b}]\ \frac{\text{Id} \neq \text{Id}'}{\textit{lookup}\ \text{Id}\ \textit{in}\ (\text{Id}'\!:\!\text{E}) \oplus \text{T} = \textit{lookup}\ \text{Id}\ \textit{in}\ \text{T}}$$

```
end Tables
```

Equation [T1b] has a negative condition Id ≠ Id′. After binding parameter Entries, the presence of [T1b] gives rise to a large number of quasi-initial models (see Section 1) in which at least two syntactically different identifiers are semantically equal. The original initial model of module Identifiers does not persist in these models. Obviously, the intended semantics is the quasi-initial model in which identifiers retain their original meaning. This model is uniquely determined up to isomorphism. The translation to Prolog given in Section 4 corresponds to this model of Tables rather than to the other ones.

No equations are given for the error case, i.e., the *lookup* operation on the empty table. As a result,

*lookup* Id *in empty-table*

gives rise to a new element of sort ENTRY (or rather of the actual sort to which ENTRY is bound) in the preferred quasi-initial algebra for every value of Id. In other words, Tables is not *sufficiently-complete* with respect to its actual parameter [GH78]. For instance, in module Type-environments (Section 2.4.1) ENTRY is bound to sort TYPE of module Pico-types, *empty-table* is renamed to *empty-tenv*, and

*lookup* Id *in empty-tenv*

gives rise to a non-standard or improper type for every value of Id. It is important to be aware of the existence of such non-standard values as they may occasionally cause problems. In the typechecker (Section 2.4.2), non-standard types are rejected by means of the *compatible* predicate defined on Pico-types. In the evaluator (Section 2.5.2), we achieve this by making the additional assumption that only statically correct programs are evaluated.

In a partial algebra approach (Section 1),

*lookup* Id *in empty-tenv*

would be undefined for every value of Id, the initial model would not contain non-standard types, and the *compatible* predicate would be superfluous.

## 2.3 The syntax of PICO

All syntactic aspects of PICO (i.e., lexical, concrete and abstract syntax) are defined in the following specification:

```
module Pico-syntax
begin
    exports
    begin
        lexical syntax
            layout SPACE
            functions
                [ \t\n\r ]    → SPACE
        context-free syntax
            sorts   PROGRAM, DECLS, ID-TYPE, SERIES,
                STATEMENT, EXP
            functions
                begin DECLS SERIES end          → PROGRAM
                declare {ID-TYPE ","}* ";"      → DECLS
                ID ":" TYPE                     → ID-TYPE
                {STATEMENT ";"}*                → SERIES

                ID ":=" EXP                     → STATEMENT
                if EXP then SERIES else SERIES fi → STATEMENT
                while EXP do SERIES od          → STATEMENT

                EXP "+" EXP                     → EXP {par, assoc}
                EXP "−" EXP                     → EXP {par}
                EXP "||" EXP                    → EXP {par, assoc}
                ID                              → EXP
                NAT-CON                         → EXP
                STR-CON                         → EXP
    end

    imports
        Identifiers, Naturals, Strings, Pico-types
end Pico-syntax
```

## 2.4 Typechecking

### 2.4.1 Type environments

First, we introduce type-environments, i.e., mappings from identifiers to types. They are an instantiation of module Tables.

```
module Type-environments
begin
   imports
      Tables
         Entries bound by
            sorts
               ENTRY ⇒ TYPE
         to Pico-types
         renamed by
            sorts
               PAIR   ⇒ TPAIR
               TABLE ⇒ TENV
            functions
               empty-table → TABLE ⇒ empty-tenv → TENV
         end renaming
end Type-environments
```

### 2.4.2 Typechecking of PICO programs

Next, typechecking of PICO programs is defined as follows:

```
module Pico-typecheck
begin
   exports
   begin
      context-free syntax
         functions
            "[" PROGRAM "]"           → BOOL
            "[" DECLS "]"             → TENV
            "[" SERIES "]" TENV       → BOOL
            "[" STATEMENT "]" TENV    → BOOL
            "[" EXP "]" TENV          → TYPE
   end
   imports
      Booleans, Pico-syntax, Type-environments
   variables
      D               → DECLS
      S, S1, S2       → SERIES
      E               → TENV
      Id              → ID
      Type            → TYPE
      Id-type-list    → {ID-TYPE ","}*
      Stat            → STATEMENT
      Stat-list       → {STATEMENT ";"}*
      Exp, Exp1, Exp2 → EXP
      Str-con         → STR-CON
      Nat-con         → NAT-CON
```

**equations**

[Tc1a] $$\frac{[\mathrm{S}]_{[\mathrm{D}]} = \mathit{true}}{[\mathit{begin}\ \mathrm{D}\ \mathrm{S}\ \mathit{end}] = \mathit{true}}$$

[Tc1b] $$\frac{[\mathrm{S}]_{[\mathrm{D}]} \neq \mathit{true}}{[\mathit{begin}\ \mathrm{D}\ \mathrm{S}\ \mathit{end}] = \mathit{false}}$$

[Tc2] $[\mathit{declare}\ \mathrm{Id}:\mathrm{Type},\ \mathrm{Id\text{-}type\text{-}list}\,;] =$
$(\mathrm{Id}:\mathrm{Type}) \oplus [\mathit{declare}\ \mathrm{Id\text{-}type\text{-}list}\,;]$

[Tc3] $[\mathit{declare}\ ;] = \mathit{empty\text{-}tenv}$

[Tc4] $$\frac{[\mathrm{Stat}]_{\mathrm{E}} = \mathit{true},\ \ [\mathrm{Stat\text{-}list}]_{\mathrm{E}} = \mathit{true}}{[\mathrm{Stat}\,;\,\mathrm{Stat\text{-}list}]_{\mathrm{E}} = \mathit{true}}$$

[Tc5] $[]_{\mathrm{E}} = \mathit{true}$

[Tc6] $$\frac{\mathit{compatible}([\mathrm{Id}]_{\mathrm{E}},\ [\mathrm{Exp}]_{\mathrm{E}}) = \mathit{true}}{[\mathrm{Id} := \mathrm{Exp}]_{\mathrm{E}} = \mathit{true}}$$

[Tc7] $$\frac{[\mathrm{Exp}]_{\mathrm{E}} = \mathit{natural},\ \ [\mathrm{S}_1]_{\mathrm{E}} = \mathit{true},\ \ [\mathrm{S}_2]_{\mathrm{E}} = \mathit{true}}{[\mathit{if}\,\mathrm{Exp}\ \mathit{then}\ \mathrm{S}_1\ \mathit{else}\ \mathrm{S}_2\ \mathit{fi}]_{\mathrm{E}} = \mathit{true}}$$

[Tc8] $$\frac{[\mathrm{Exp}]_{\mathrm{E}} = \mathit{natural},\ \ [\mathrm{S}]_{\mathrm{E}} = \mathit{true}}{[\mathit{while}\ \mathrm{Exp}\ \mathit{do}\ \mathrm{S}\ \mathit{od}]_{\mathrm{E}} = \mathit{true}}$$

[Tc9] $$\frac{[\mathrm{Exp}_1]_{\mathrm{E}} = \mathit{natural},\ \ [\mathrm{Exp}_2]_{\mathrm{E}} = \mathit{natural}}{[\mathrm{Exp}_1 + \mathrm{Exp}_2]_{\mathrm{E}} = \mathit{natural}}$$

[Tc10] $$\frac{[\mathrm{Exp}_1]_{\mathrm{E}} = \mathit{natural},\ \ [\mathrm{Exp}_2]_{\mathrm{E}} = \mathit{natural}}{[\mathrm{Exp}_1 - \mathrm{Exp}_2]_{\mathrm{E}} = \mathit{natural}}$$

[Tc11] $$\frac{[\mathrm{Exp}_1]_{\mathrm{E}} = \mathit{string},\ \ [\mathrm{Exp}_2]_{\mathrm{E}} = \mathit{string}}{[\mathrm{Exp}_1\ ||\ \mathrm{Exp}_2]_{\mathrm{E}} = \mathit{string}}$$

[Tc12] $[\mathrm{Nat\text{-}con}]_{\mathrm{E}} = \mathit{natural}$

[Tc13] $[\mathrm{Str\text{-}con}]_{\mathrm{E}} = \mathit{string}$

[Tc14] $[\mathrm{Id}]_{\mathrm{E}} = \mathit{lookup}\ \mathrm{Id}\ \mathit{in}\ \mathrm{E}$

**end** Pico-typecheck

Like [T1b] in Section 2.2.3, the negative conditional equation [Tc1b] gives rise to a large number of undesirable quasi-initial models. In most of these models there is at least one program *p* such that

$$[p] = \mathit{true}$$

without this being equationally provable from the axioms [Tc1a] and [Tc2]-[Tc14]. The model we have in mind, however, contains no such *p*. This requirement characterizes the preferred quasi-initial model uniquely up to isomorphism.

Rather than in the form given, we could have written [Tc6] in the simpler form

$$\frac{[\mathrm{P}] \neq \mathit{true}}{[\mathrm{P}] = \mathit{false}}$$

with P a variable of sort PROGRAM. In this form, however, the equation is not *reducing* (*simplifying*) in the sense of Kaplan [Kap87a, Kap87b]. The condition $[\mathrm{P}] \neq \mathit{true}$ is not in any sense syntactically simpler than the conclusion $[\mathrm{P}] = \mathit{false}$, and the corresponding conditional rewrite rule causes termination problems. We have avoided non-reducing equations in our specification whenever possible.

Equation [Tc6] is the only one using the *compatible* predicate discussed in Section 2.2.3. If the condition in [Tc6] would have been written in the form

$$[\text{Id}]_E \;=\; [\text{Exp}]_E$$

the PICO assignment

$$x := x$$

with undeclared PICO variable *x* would become correct, as this condition is clearly satisfied with

$$[\text{Id}]_E \;=\; [\text{Exp}]_E \;=\; \textit{lookup } x \textit{ in empty-tenv}.$$

With the condition

$$\textit{compatible}([\text{Id}]_E,[\text{Exp}]_E) \;=\; \textit{true}$$

only standard types are accepted.

As was pointed out in Section 2.2.3, the natural partial algebra defined by Pico-typecheck does not contain non-standard types. Hence, the *compatible* predicate is not needed and [Tc6] can simply be written with condition

$$[\text{Id}]_E \;=\; [\text{Exp}]_E.$$

## 2.5 Evaluation

First, we introduce in Section 2.5.1 the notions of Pico-values and Value-environments. Next, evaluation of PICO programs is defined in Section 2.5.2.

### 2.5.1 Values and value environments

The only values that may occur during the evaluation of PICO programs are natural numbers and strings. We embed both in the sort VAL, as can be seen in the following module Pico-values:

```
module Pico-values
begin

   exports
   begin
      context-free syntax
         sorts    VAL
         functions
            NAT      → VAL
            STRING → VAL
   end

   imports
      Naturals, Strings

end Pico-values
```

Value-environments are mappings from identifiers to values and are obtained by instantiation of module Tables:

```
module Value-environments
begin

   imports
      Tables
         Entries bound by
            sorts
               ENTRY ⇒ VAL
```

```
        to Pico-values
        renamed by
           sorts
              PAIR    ⇒ VPAIR
              TABLE  ⇒ VENV
           functions
              empty-table → TABLE  ⇒  empty-venv → VENV
        end renaming
end Value-environments
```

**2.5.2 Evaluation of PICO programs**

Finally, we specify evaluation of PICO programs. In this specification we make the assumption that only *statically correct* programs are evaluated.

```
module Pico-eval
begin

   exports
   begin
      context-free syntax
         functions
            "[" PROGRAM "]"              → VENV
            "[" DECLS "]"                → VENV
            "[" SERIES "]" VENV          → VENV
            "[" STATEMENT "]" VENV       → VENV
            "[" EXP "]" VENV             → VAL
   end

   imports
      Booleans, Pico-syntax, Value-environments

   variables
      D                    → DECLS
      S, S1, S2            → SERIES
      E                    → VENV
      Id                   → ID
      Id-type-list         → {ID-TYPE ","}*
      Stat                 → STATEMENT
      Stat-list            → {STATEMENT ";"}*
      Exp, Exp1, Exp2      → EXP
      Str-con              → STR-CON
      Nat-con              → NAT-CON
      Val                  → VAL
      Nat1, Nat2           → NAT
      Str1 , Str2          → STRING

   equations
      [Ev1]    [begin D S end] = [S][D]

      [Ev2a]   [declare Id : natural, Id-type-list ;] =
                  (Id : 0) ⊕ [declare Id-type-list ;]

      [Ev2b]   [declare Id : string, Id-type-list ;] =
                  (Id : "") ⊕ [declare Id-type-list ;]

      [Ev3]    [declare ;] = empty-venv

      [Ev4]    [Stat ; Stat-list]E = [Stat-list][Stat]E
```

[Ev5] $[]_E = E$

[Ev6] $[\mathrm{Id} := \mathrm{Exp}]_E = (\mathrm{Id} : [\mathrm{Exp}]_E) \oplus E$

[Ev7a] $$\frac{[\mathrm{Exp}]_E = 0}{[\textit{if}\ \mathrm{Exp}\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\,]_E = [S_2]_E}$$

[Ev7b] $$\frac{[\mathrm{Exp}]_E \neq 0}{[\textit{if}\ \mathrm{Exp}\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\,]_E = [S_1]_E}$$

[Ev8a] $$\frac{[\mathrm{Exp}]_E = 0}{[\textit{while}\ \mathrm{Exp}\ \textit{do}\ S\ \textit{od}\,]_E = E}$$

[Ev8b] $$\frac{[\mathrm{Exp}]_E \neq 0}{[\textit{while}\ \mathrm{Exp}\ \textit{do}\ S\ \textit{od}\,]_E = [\textit{while}\ \mathrm{Exp}\ \textit{do}\ S\ \textit{od}\,]_{[S]_E}}$$

[Ev9] $$\frac{[\mathrm{Exp}_1]_E = \mathrm{Nat}_1,\ \ [\mathrm{Exp}_2]_E = \mathrm{Nat}_2}{[\mathrm{Exp}_1 + \mathrm{Exp}_2]_E = \mathrm{Nat}_1 + \mathrm{Nat}_2}$$

[Ev10] $$\frac{[\mathrm{Exp}_1]_E = \mathrm{Nat}_1,\ \ [\mathrm{Exp}_2]_E = \mathrm{Nat}_2}{[\mathrm{Exp}_1 - \mathrm{Exp}_2]_E = \mathrm{Nat}_1 - \mathrm{Nat}_2}$$

[Ev11] $$\frac{[\mathrm{Exp}_1]_E = \mathrm{Str}_1,\ \ [\mathrm{Exp}_2]_E = \mathrm{Str}_2}{[\mathrm{Exp}_1 \mid\mid \mathrm{Exp}_2]_E = \mathrm{Str}_1 \mid\mid \mathrm{Str}_2}$$

[Ev12] $[\text{Nat-con}]_E = \text{Nat-con}$

[Ev13] $[\text{Str-con}]_E = \text{Str-con}$

[Ev14] $[\mathrm{Id}]_E = \textit{lookup}\ \mathrm{Id}\ \textit{in}\ E$

**end** Pico-eval

## 3 IMPLEMENTATION ISSUES

We will briefly sketch a method for implementing the specification presented in this paper. Our method is an extension of the "compilational" approach to the translation of algebraic specifications to Prolog programs by van Emden and Yukawa [EY87]. This method takes a set of equations $E$ as input and produces an equivalent Prolog program $P$. This translation is obtained by:

- Replacing each $n$-ary function symbol $f$ by a corresponding $n+1$-ary predicate $F$ such that

$$\forall x_1, \ldots, x_{n+1}\ f(x_1, \ldots, x_n) = x_{n+1} \Leftrightarrow F(x_1, \ldots, x_n, x_{n+1});$$

- Eliminating nested function applications by introducing auxiliary variables and extra conditions.

The conditions originally imposed by van Emden and Yukawa are:

- The set $E$ should correspond to a finitely terminating term rewriting system.
- Each equation should be of the form $f(t_1, \ldots, t_n) = t$, where the $t_i$ may only contain *constructors*, i.e., functions that do not occur at the outermost position in the left-hand side of any equation. This is a purely syntactic requirement.
- All normal forms of ground terms must consist solely of constructors. This is a semantic requirement which may be difficult to verify. (Usually, any function occurring in a normal form is called a constructor but the "syntactical" constructors of van Emden and Yukawa do not necessarily have this property.)

We extend van Emden and Yukawa's scheme in the following ways:

- We also apply it to positive/negative conditional equations. Negative conditions are implemented by means of the Prolog inequality operator \==.
- We also apply it to non-terminating systems In this particular specification non-termination may be caused by [Ev8], the rule for *while*-statements. All other equations are *reducing* (Section 2.4.2) and do

not present termination problems.

- We do not require normal forms to consist solely of constructors, as this would not work in our case. For instance, normal forms of sort BOOL may contain *defined functions* (i.e., functions that are not constructors in the sense of van Emden and Yukawa).
- We map lists of items (as defined by * and + in the SDF definition) directly onto Prolog lists.

As an example, consider the Prolog translation of equations [Tc4] and [Tc5], which define the typechecking of a series of statements (see Section 2.4.1):

```
/* Tc4 */ tcsr(series([Stat | StatList]), E, true) :-
              tcst(Stat, E, true),
              tcsr(series(StatList), E, true), !.
/* Tc5 */ tcsr(series([]), E, true).
/* --- */ tcsr(S, E, tcsr(S, E)).
```

In this (hand) translation we adopt a straightforward convention for naming the constructor functions of the abstract syntax of PICO (`program`, `decls`, `series`, `assign`, etc.) and for distinguishing the overloaded versions of the typechecking and evaluation function (these names begin with the prefix `tc` and `ev`, respectively).

The translation of each *defined* function in the specification consists of the translation of all conditional equations in whose left-hand side the function occurs as outermost symbol, followed by a "catch-all" clause of the form `f(T1,...,Tn, f(T1,...,Tn))`. This clause models the "omitted equations" for function `f` and returns the original term `f(T1,...,Tn)` as result. Note that `f` is overloaded in the catch-all clause: the first occurrence is the $n$+1-ary predicate resulting from the translation of function `f` in the specification; the second occurrence is an $n$-ary constructor function representing a normal form. Compare this technique with [CM84], p. 175. Constructors are translated as before using Prolog terms without corresponding predicate and hence without catch-all clause.

We use auxiliary Prolog predicates for operations on natural numbers (`add` and `csub`) and strings (`conc`). Their definitions are not shown.

The precise conditions under which the translation method used in this paper is correct still have to be formulated

## 4 EXECUTABLE VERSION OF THE PICO SPECIFICATION

Although a compiler has been implemented using the translation scheme of the previous section, we give a hand translation of the PICO specification for reasons of readability. Identifiers in the original specification have been retained in the translation as much as possible.

The translation is:

```
/* Module Pico-types ---------------------------------*/
/* Typ1 */ compatible(natural, natural, true).
/* Typ2 */ compatible(string, string, true).
/* ---- */ compatible(X, Y, compatible(X, Y)).

/* Module Type-environments --------------------------*/
/* T1a  */ lookup(Id, '.'(':'(Id,E), T), E).
/* T1b  */ lookup(Id, '.'(':'(Id1,E), T), E1) :-
               Id \== Id1,
               lookup(Id, T, E1), !.
/* ---- */ lookup(Id, T, lookup(Id, T)).

/* Module Pico-typecheck -----------------------------*/
/* Tc1a */ tcp(program(D,S), true) :-
               tcd(D, E), tcsr(S, E, true), !.
/* Tc1b */ tcp(program(D,S), false) :-
               tcd(D, E), tcsr(S, E, B), B \== true, !.
/* ---- */ tcp(P, tcp(P)).
/* Tc2  */ tcd(decls([id_type(Id,Type)|IdTypeList]),
               '.'(':'(Id, Type), E)) :-
               tcd(decls(IdTypeList), E).
```

```
/* Tc3  */ tcd(decls([]), empty_tenv).
/* ---- */ tcd(D, tcd(D)).
/* Tc4  */ tcsr(series([Stat | StatList]), E, true) :-
                tcst(Stat,E, true),
                tcsr(series(StatList),E, true), !.
/* Tc5  */ tcsr(series([]), E, true).
/* ---- */ tcsr(S, E, tcsr(S, E)).
/* Tc6  */ tcst(assign(Id, Exp), E, true) :-
                tce(Id, E, Type1), tce(Exp, E, Type2),
                compatible(Type1, Type2, true), !.
/* Tc7  */ tcst(if(Exp,S1,S2), E, true) :-
                tce(Exp, E, natural),
                tcsr(S1, E, true),
                tcsr(S2, E, true), !.
/* Tc8  */ tcst(while(Exp,S), E, true) :-
                tce(Exp, E, natural),
                tcsr(S, E, true), !.
/* ---- */ tcst(S, E, tcst(S, E)).
/* Tc9  */ tce('+'(Exp1,Exp2), E, natural) :-
                tce(Exp1, E, natural),
                tce(Exp2, E, natural), !.
/* Tc10 */ tce('-'(Exp1,Exp2), E, natural) :-
                tce(Exp1, E, natural),
                tce(Exp2, E, natural), !.
/* Tc11 */ tce('||'(Exp1,Exp2), E, string) :-
                tce(Exp1, E, string),
                tce(Exp2, E, string), !.
/* Tc12 */ tce(nat_con, E, natural).
/* Tc13 */ tce(str_con, E, string).
/* Tc14 */ tce(Id, E, Type) :-
                lookup(Id, E, Type), !.
/* ---- */ tce(Exp, E, tce(Exp, E)).

/* Module Value-environments ------------------------*/
/* T1a  */ lookup(Id, '.'(':'(Id,Val), T), Val).
/* T1b  */ lookup(Id, '.'(':'(Id1,Val), T), Val1) :-
                Id \== Id1,
                lookup(Id, T, Val1), !.
/* ---- */ lookup(Id, T, lookup(Id, T)).

/* Module Pico-eval ----------------------------------*/
/* Ev1  */ evp(program(D,S), E1) :-
                evd(D, E),
                evsr(S, E, E1), !.
/* ---- */ evp(P, evp(P)).
/* Ev2a */ evd(decls([id_type(Id,natural)|IdTypeList]),
               '.'(':'(Id, val(0)), E1)) :-
                evd(decls(IdTypeList), E1), !.
/* Ev2b */ evd(decls([id_type(Id,string)|IdTypeList]),
               '.'(':'(Id, val([])), E1)) :-
                evd(decls(IdTypeList), E1), !.
/* Ev3  */ evd(decls([]), empty_venv).
/* ---- */ evd(D, evd(D)).
/* Ev4  */ evsr(series([Stat | StatList]), E, E2) :-
                evst(Stat, E, E1),
                evsr(series(StatList), E1, E2), !.
/* Ev5  */ evsr(series([]), E, E).
/* ---- */ evsr(S, evsr(S)).
/* Ev6  */ evst(assign(Id, Exp), E,
               '.'(':'(Id, Val), E)) :-
                eve(Exp, E, Val), !.
/* Ev7a */ evst(if(Exp,S1,S2), E, E1) :-
                eve(Exp, E, Val),
                Val == val(0),
                evsr(S2, E, E1), !.
```

```
/* Ev7b */ evst(if(Exp,S1,S2), E, E1) :-
                eve(Exp, E, Val),
                Val \== val(0),
                evsr(S1, E, E1), !.
/* Ev8a */ evst(while(Exp,S), E, E) :-
                eve(Exp, E, Val),
                Val  == val(0), !.
/* Ev8b */ evst(while(Exp,S), E, E2) :-
                eve(Exp, E, Val),
                Val  \== val(0),
                evsr(S, E, E1),
                evst(while(Exp,S), E1, E2), !.
/* ---- */ evst(S, E, evst(S, E)).
/* Ev9  */ eve('+'(Exp1,Exp2), E, val(Nat3)) :-
                eve(Exp1, E, val(Nat1)),
                eve(Exp2, E, val(Nat2)),
                add(Nat1, Nat2, Nat3), !.
/* Ev10 */ eve('-'(Exp1,Exp2), E, val(Nat3)) :-
                eve(Exp1, E, val(Nat1)),
                eve(Exp2, E, val(Nat2)),
                csub(Nat1, Nat2, Nat3), !.
/* Ev11 */ eve('||'(Exp1,Exp2), E, val(Str3)) :-
                eve(Exp1, E, val(Str1)),
                eve(Exp2, E, val(Str2)),
                conc(Str1, Str2, Str3), !.
/* Ev12 */ eve(nat_con(Nat), E, val(Nat)).
/* Ev13 */ eve(str_con(Str), E, val(Str)).
/* Ev14 */ eve(Id, E, V) :- lookup(Id, E, V).
/* ---- */ eve(Exp, E, eve(Exp, E)).
```

## 5 CONCLUSIONS AND FURTHER WORK

The aim of the specification style introduced in this paper was to avoid the specification of "uninteresting" cases. This style clearly gives concise specifications, but there remain subtle points one should be aware of such as, for instance, the naive (and erroneous) formulation of [Tc6] we encountered in Section 2.4.2. Such problems seem to be difficult to avoid when specifying partial functions in a formalism that only allows total ones. In retrospect, we believe that a partial algebra interpretation would have been the most natural one in our case. Rather than needing catch-all clauses for computing non-standard values, the corresponding Prolog translation would use standard Prolog failure propagation to model propagation of undefined values.

The advantages of using negative conditions will be apparent from the specification in this paper. Without them, it would have been necessary to define equality predicates for sorts ID and VAL for comparing identifiers in the lookup operation on tables and values in the evaluation of *if*- and *while*-statements. It would also have been difficult to define the typechecking of incorrect programs to be equal to *false* as is done in equation [Tc1b].

## ACKNOWLEDGEMENTS

Dominique Clément, Thierry Despeyroux and Gilles Kahn stimulated us to think about abbreviating our specifications by asking why we needed to specify all *false* cases of Boolean functions. Paul Hendriks suggested the use of "catch-all" clauses discussed in Section 3. He also implemented the translation scheme discussed there. Freek Wiedijk identified the problem of "unintended" equalities between non-standard values discussed in Section 2.4.2.

## REFERENCES

[BHK85] J.A. Bergstra, J. Heering & P. Klint, "Algebraic definition of a simple programming language", Report CS-R8504, Centre for Mathematics and Computer Science, 1985.

[BHK87] J.A. Bergstra, J. Heering & P. Klint, "ASF - An algebraic specification formalism", Report CS-R8705, Centre for Mathematics and Computer Science, 1987.

[CM84] W.F. Clocksin & C.S. Mellish, *Programming in Prolog*, 2nd. ed., Springer-Verlag, 1984.

[Des84] T. Despeyroux, "Executable specification of semantics", *Semantics of Data Types*, ed. G. Kahn, D.B. MacQueen & G. Plotkin, Lecture Notes in Computer Science, vol. 173, pp. 215-233, Springer-Verlag, 1984.

[EY87] M.H. van Emden & K. Yukawa, "Logic programming with equations", *Journal of Logic Programming*, vol. 4, pp. 265-288, 1987.

[GH78] J.V. Guttag & J.J. Horning, "The algebraic specification of abstract datatypes", *Acta Informatica*, vol. 10, pp. 27-52, 1978.

[Hen87] P.R.H. Hendriks, "Type-checking Mini-ML: an algebraic specification with user-defined syntax", Report CS-R8737, Centre for Mathematics and Computer Science, Amsterdam, 1987. Abbreviated version in *Proceedings of CSN87: Computing Science in the Netherlands*, pp. 21-38, SION, 1987.

[HK87] J. Heering & P. Klint, "A syntax definition formalism", in ESPRIT'86: Results and Achievements, North-Holland, 1987, pp. 619-630.

[GJM85] J.A. Goguen, J.-P. Jouannoud & J. Meseguer, "Operational semantics of order-sorted algebra", in *Automata, Languages and Programming*, ed. W. Brauer, Lecture Notes in Computer Science, vol. 194, pp. 221-231, Springer-Verlag, 1985.

[GM85] J.A. Goguen & J. Meseguer, "Order-sorted algebra I: Partial and overloaded operators, errors and inheritance", Preprint, 1985.

[Kah87] G. Kahn, "Natural semantics", in *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, ed. F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, Lecture Notes in Computer Science, vol. 247, pp. 22-39, Springer-Verlag, 1987.

[Kap87a] S. Kaplan, "Simplifying conditional term rewriting systems: unification, termination and confluence", *Journal of Symbolic Computation*, vol. 4, pp. 295-334, 1987.

[Kap87b] S. Kaplan, "Positive/negative conditional rewriting", Technical report 87-10, Leibniz Center for Research in Computer Science, Hebrew University, Jerusalem, 1987.

[L84] J.W. Lloyd, *Foundations of Logic programming*, Springer-Verlag, 1984.

[Plo81] G.D. Plotkin, "A structural approach to operational semantics", Report DAIMI FN-19, Aarhus University, 1987.

[Rei87] H. Reichel, *Initial Computability, Algebraic Specifications, and Partial Algebras*, Clarendon Press, Oxford, 1987.