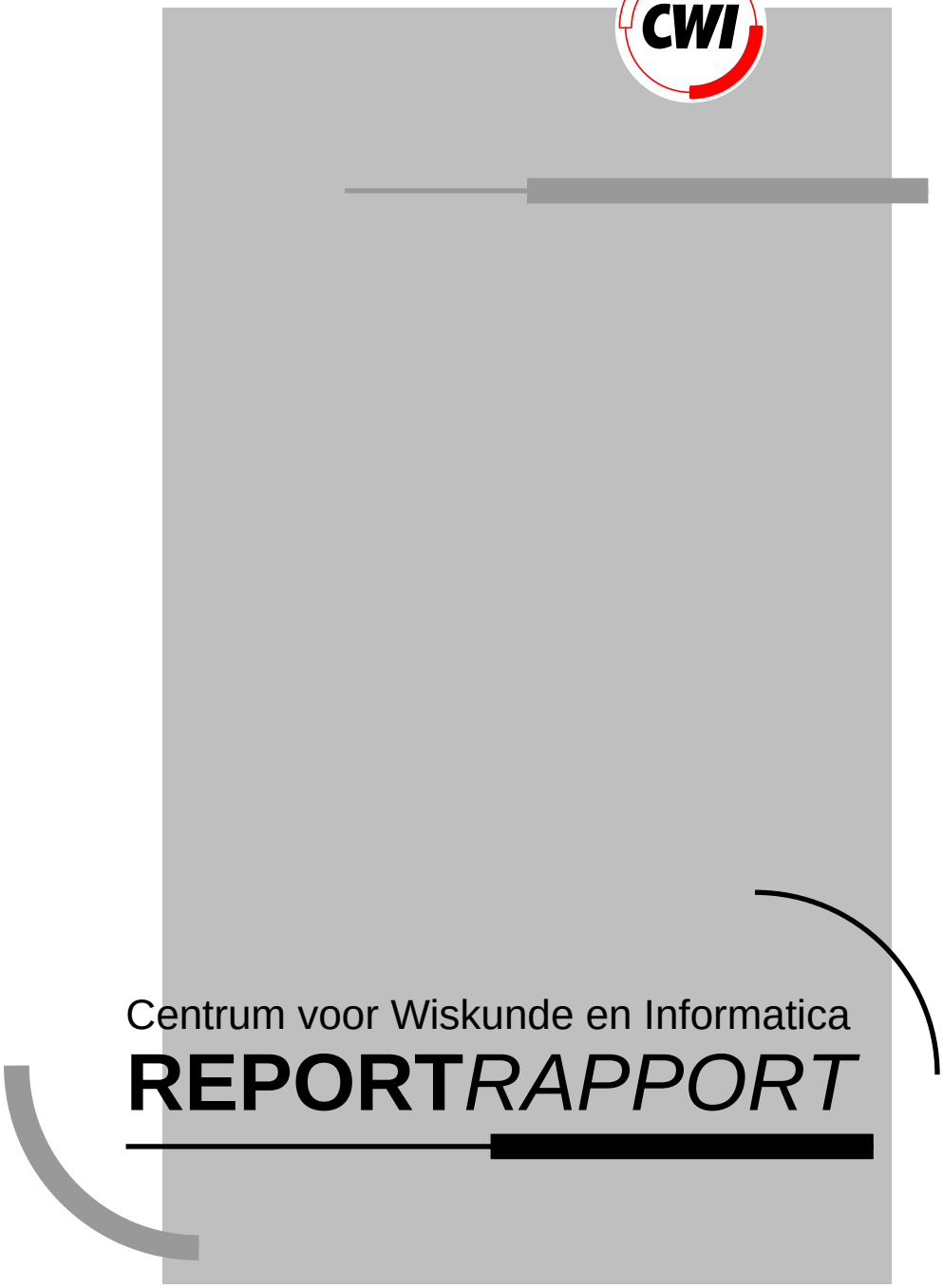

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Second-order algebraic specification of static semantics

J. Heering

Computer Science/Department of Software Technology

CS-R9254 1992

Second-Order Algebraic Specification of Static Semantics

Jan Heering
CWI
P.O. Box 4079, 1009 AB Amsterdam
The Netherlands
jan@cwi.nl

Abstract

Higher-order algebraic specification is a synthesis of first-order algebraic specification and higher-order functional programming which is both logically appealing as well as considerably more expressive than its two predecessors. To illustrate this, we describe the static semantics of a simple block-structured programming language using second-order equations in addition to first-order ones. The specification has a highly non-deterministic character and does not use a type environment. Furthermore, it supports error recovery and early detection of errors in incomplete programs.

1991 Mathematics Subject Classification: 68Q65 [**Theory of computing]:** Abstract data types; algebraic specification.

1991 CR Categories: F.3.2 [**Logics and meanings of programs]:** Semantics of programming languages - *Algebraic approaches to semantics.*

Key Words & Phrases: higher-order algebraic specification, static semantics, non-deterministic specification.

Note: Supported in part by the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II—GIPE II).

1 Introduction

1.1 Higher-order algebraic specification

Higher-order algebraic specification (Parsaye-Ghomi [22], Poigné [24], Möller [18], Jouannaud and Okada [14], Meinke [16], Heering [8]) is a synthesis of first-order algebraic specification and higher-order functional programming which is both logically appealing as well as considerably more expressive than its two predecessors. It is obtained by allowing both signatures and equations in algebraic specifications to be higher-order. The higher-order signature defines the abstract syntax of a language of typed λ-terms whose semantics is given by the equations of the specification, in conjunction with the built-in rules of α-, β- and η-conversion for typed λ-terms.

The implementation of higher-order algebraic specifications in terms of rewrite rules uses *higher-order matching* (matching of typed λ-terms) as its basic computational primitive. Higher-order matching is the special case of higher-order unification in which only one of the terms involved is instantiated. Since higher-order unification is available in the generic theorem prover Isabelle [21] as well as in the λProlog programming language [19], it is not difficult to perform operational experiments with higher-order algebraic specifications. The specifications in this paper were implemented in λProlog using the translation schemes outlined in [8].

Second-order matching (which is all we need in this paper) is discussed by Huet and Lang [11], and, in a more general setting, by Dowek [5]. A good reference on higher-order unification is the recent book by Snyder [25]. We note that second-order matching is decidable, whereas second-order unification is not.

Alternatively, an implementation of higher-order algebraic specifications could be based on *higher-order narrowing*, which is a combination of higher-order unification and rewriting similar to first-order narrowing. Such an implementation would be a higher-order analogue of a system like RAP [12], which is a first-order algebraic specification system based on narrowing. Although the above-mentioned λProlog schemes support higher-order narrowing *in principle*, narrowing requires much tighter control than rewriting to be effective. We have not yet performed serious experiments in this direction.

1.2 The higher-order algebraic specification formalism

We use a higher-order extension of the ASF+SDF language, a first-order algebraic specification formalism with general user-definable syntax [2, 7, 10]. Its basic features will be explained in due course, so the reader need not be familiar with it. Suffice it to say at this point that the right- and left-hand sides of equations are written in the concrete syntax defined in the declarations part (signature) of the specification. While a highly incremental language definition system based on ASF+SDF is available [15], we do not yet have an implementation of the full higher-order extension of ASF+SDF. The λProlog implementation route mentioned in Section 1.1 does not handle user-definable syntax.

1.3 Some issues in the specification of static semantics

Our point of departure is the observation that most first-order static semantics specifications are overly deterministic. The declaration and use of an identifier in a program can be related only by an explicit traversal of the program, the declaration information being carried along in the type environment. In view of this, our first two desiderata for static semantics specifications are:

- No explicit program traversal.
- No type environment.

Since it is to be expected that execution of a specification in this style is not hampered by incorrect or missing parts in the program being checked, we mention another desirable property:

- Early detection of errors in incomplete programs containing editor place holders (cf. Bahlke and Snelting [1, Section 4.1]).

Keeping these issues in mind, we improve on an earlier specification in [2, Chapter 9], and construct a second-order algebraic specification for the static semantics of the toy language Pico in Section 2. Block structure will be added in Section 3. The use of higher-order abstract syntax (typed λ-term representation) for blocks will enable us to extend the original specification in a modular way.

A first-order static semantics specification that does not use a type environment (although for a different reason) has been discussed by Walters [26].

2 Static semantics of Pico

2.1 The syntax of Pico

The following ASF+SDF module defines the concrete and (first-order) abstract syntax of Pico (cf. [2, Chapter 9]):

module Pico-syntax
imports Layout Identifiers Naturals Strings
sorts PROGRAM DECLS DECL SERIES STATEMENT EXPR TYPE
functions begin DECLS SERIES end → PROGRAM
declare {DECL , }* ; → DECLS
ID : TYPE → DECL
{STATEMENT ; }* → SERIES
ID := EXPR → STATEMENT
if EXPR then SERIES else SERIES fi → STATEMENT
while EXPR do SERIES od → STATEMENT
EXPR + EXPR → EXPR {assoc}
EXPR − EXPR → EXPR {left-assoc}
EXPR ‖ EXPR → EXPR {assoc}
ID → EXPR
NATURAL → EXPR
STRING → EXPR
(EXPR) → EXPR {bracket}
natural → TYPE
string → TYPE

From the viewpoint of concrete syntax, sorts correspond to non-terminals and a function declaration is a BNF syntax rule written *in reverse*. The corresponding abstract syntax is obtained by reading the function declarations in the usual way from left to right and ignoring the distfix character of function names. Associative (flat) lists with elements of sort S and concrete list separator d are denoted by {S d}*. These may be decomposed by means of patterns containing list variables of type {S d}* in addition to other elements. Associative lists and list variables are a built-in feature of ASF+SDF [10, Chapter 3]. The sorts ID, NATURAL, and STRING occurring in the signature are imported from modules Identifiers, Naturals, and Strings respectively. These are not shown.

2.2 Type checking—the basic case

The basic static semantics of Pico requires only 9 equations, all of them first-order except the first one:

module Pico-typecheck
imports Pico-syntax (Section 2.1)
functions tp(TYPE) → ID
variables *Series*(ID) → SERIES
D_1, D_2 → {DECL , }*
S, S', S_1, S_2 → {STATEMENT ; }*
X → ID
τ → TYPE
n → NATURAL
σ → STRING

equations

$$\begin{array}{rcl} \begin{array}{l}\text{begin}\\ \text{declare } D_1, X:\tau, D_2;\\ \textit{Series}(X)\\ \text{end}\end{array} & = & \begin{array}{l}\text{begin}\\ \text{declare } D_1, X:\tau, D_2;\\ \textit{Series}(\text{tp}(\tau))\\ \text{end}\end{array} \qquad (1)\\ n & = & \text{tp(natural)} \qquad (2)\\ \sigma & = & \text{tp(string)} \qquad (3)\\ S_1; \text{tp}(\tau) := \text{tp}(\tau); S_2 & = & S_1; S_2 \qquad (4)\\ S_1; \text{if tp(natural) then } S \text{ else } S' \text{ fi}; S_2 & = & S_1; S; S'; S_2 \qquad (5)\\ S_1; \text{while tp(natural) do } S \text{ od}; S_2 & = & S_1; S; S_2 \qquad (6)\\ \text{tp(natural)} + \text{tp(natural)} & = & \text{tp(natural)} \qquad (7)\\ \text{tp(natural)} - \text{tp(natural)} & = & \text{tp(natural)} \qquad (8)\\ \text{tp(string)} \parallel \text{tp(string)} & = & \text{tp(string)} \qquad (9) \end{array}$$

Equation (1) uses the second-order variable *Series*. According to its declaration in the **variables**-section, it is of type ID → SERIES. The variables D_1 and D_2 are list variables of type {DECL , }* (see Section 2.1). Viewed as a left-to-right rewrite rule, equation (1) picks up a declaration $X : \tau$ in the list of declarations by means of associative list matching, and an arbitrary number (possibly 0—see below) of occurrences of X in the statements section by means of second-order matching. It then replaces the occurrences

of X (if any) with τ, or rather with the corresponding identifier $\mathrm{tp}(\tau)$. The character of second-order variables is such that an identifier X can be picked up in the statements section *regardless of the depth at wich it occurs.*

For instance, let P be the (statically incorrect) Pico program

begin
 declare a : natural, b : string;
 while a do $b := a$ od
end.

The left-hand side of equation (1) matches P in 6 different ways. Apart from the values of the list variables D_1 and D_2, the solutions are

$X = a \quad \tau = \text{natural} \quad Series = \lambda U.$ while U do $b := U$ od
$X = a \quad \tau = \text{natural} \quad Series = \lambda U.$ while U do $b := a$ od
$X = a \quad \tau = \text{natural} \quad Series = \lambda U.$ while a do $b := U$ od
$X = a \quad \tau = \text{natural} \quad Series = \lambda U.$ while a do $b := a$ od

$X = b \quad \tau = \text{string} \quad Series = \lambda U.$ while a do $U := a$ od
$X = b \quad \tau = \text{string} \quad Series = \lambda U.$ while a do $b := a$ od,

where U is a bound variable of sort ID. Hence, after a single application of (1) the possible results are

begin
 declare a : natural, b : string;
 while tp(natural) do b := tp(natural) od
end

begin
 declare a : natural, b : string;
 while tp(natural) do $b := a$ od
end

begin
 declare a : natural, b : string;
 while a do b := tp(natural) od
end

begin
 declare a : natural, b : string;
 while a do $b := a$ od
end

begin
 declare a : natural, b : string;
 while a do tp(string) := a od
end

begin
 declare a : natural, b : string;
 while a do $b := a$ od
end

The second-order matching strategy actually used does not matter, but the two solutions with

$$Series = \lambda U.\ \text{while } a \text{ do } b := a \text{ od}$$

do not pick up an identifier in the statements section. Although they are algebraically harmless, they cause a loop in the rewrite implementation which has to be suppressed. The λProlog implementation schemes described in [8] will take care of this.

We note that the left-hand side of (1) is not a higher-order pattern (HOP) in the sense of Nipkow [20] and Miller [17], since the argument X of $Series$ is not (η-equivalent to) a bound variable. Matching of HOPs resembles first-order matching in that it is unitary, a property not shared by the left-hand side of (1) as the above example shows. For this reason, it is important to make sure that the matching strategy does not influence the result.

Now, returning to Pico-typecheck, the replacement of items with their type is continued by equations (2) and (3). These apply to constants and do not need access to the declarations section of the Pico program. Note that their left- and right-hand sides are terms of sort EXPR rather than NATURAL or STRING since EXPR is the only sort that can be assigned to both sides.

The remaining equations eliminate parts of the Pico program that are statically correct. Equation (4) deletes an assignment statement whose left- and right-hand sides have the same type, and equations (5) and (6) remove the skeleton of an if- or while-statement whose expression part has the correct type tp(natural). Expressions are evaluated by equations (7)–(9).

If the program is correct, its statements section is completely eliminated by Pico-typecheck, yielding the normal form

$$\text{begin declare } \cdots; \text{ end.}$$

If it is incorrect, remnants of the offending statements will be present in the normal form. For instance, the normal form of P is

$$\begin{array}{l}\text{begin}\\ \text{ declare } a : \text{natural},\ b : \text{string};\\ \text{ tp(string)} := \text{tp(natural)}\\ \text{end},\end{array}$$

where the non-standard statement

$$\text{tp(string)} := \text{tp(natural)}$$

is a remnant of the incorrect statement $b := a$. The skeleton of the while-statement is correct and has been eliminated by (6).

Remnants of offending statements can be related to the parts of the program they originated from by a suitably extended *origin tracking facility* in the style of van Deursen, Klint, and Tip [3]. Furthermore, equations converting the normal form into a set of user-friendly diagnostic messages can be added to the specification in a straightforward way—see Dinesh and Tip [4].

The appropriate use of list variables in conjunction with second-order variables lends Pico-typecheck a highly non-deterministic character. No tree traversal is specified. As a consequence, the corresponding rewrite system not only works well on complete Pico programs but also on incomplete ones containing editor place holders (variables).

For instance, Pico-typecheck reduces

begin
declare a : natural, b : natural;
while a do < STATEMENT >; $b := a$ od
end,

where < STATEMENT > is a place holder of sort STATEMENT, to

begin
declare a : natural, b : natural;
< STATEMENT >
end.

Whether a statically correct value will be substituted for the place holder remains to be seen, so it is retained in the normal form. Everything else is eliminated.

2.3 Recovery from undeclared variables

Undeclared variables cannot be eliminated by Pico-typecheck (Section 2.2), and are left in the statements section of the normal form. In addition to these, the normal form may contain the non-standard identifiers tp(natural) and tp(string). Apart from two (important) conditions that will be added to each of the equations later on, the following module adds declarations for variables that were not declared:

module	Pico-recover	(*initial version—not yet complete*)
imports	Pico-typecheck	(Section 2.2)
variables	*Series*(STATEMENT) → SERIES	

$Series$(EXPR)	$\rightarrow$ SERIES
D	$\rightarrow$ {DECL , }*
E	$\rightarrow$ EXPR
ξ	$\rightarrow$ CHAR+
τ	$\rightarrow$ TYPE

equations

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{id}(\xi) := \text{tp}(\tau)) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \tau; \\ \text{ } Series(\text{id}(\xi) := \text{tp}(\tau)) \\ \text{end} \end{array} \quad (10)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{tp}(\tau) := \text{id}(\xi)) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \tau; \\ \text{ } Series(\text{tp}(\tau) := \text{id}(\xi)) \\ \text{end} \end{array} \quad (11)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{if id}(\xi) \text{ then } S \text{ else } S' \text{ fi}) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \text{ } Series(\text{if id}(\xi) \text{ then } S \text{ else } S' \text{ fi}) \\ \text{end} \end{array} \quad (12)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{while id}(\xi) \text{ do } S \text{ od}) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \text{ } Series(\text{while id}(\xi) \text{ do } S \text{ od}) \\ \text{end} \end{array} \quad (13)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{id}(\xi) + E) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \text{ } Series(\text{id}(\xi) + E) \\ \text{end} \end{array} \quad (14)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(E + \text{id}(\xi)) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \text{ } Series(E + \text{id}(\xi)) \\ \text{end} \end{array} \quad (15)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \text{ } Series(\text{id}(\xi) - E) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \text{ } Series(\text{id}(\xi) - E) \\ \text{end} \end{array} \quad (16)$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \ Series(E - \text{id}(\xi)) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{natural}; \\ \ Series(E - \text{id}(\xi)) \\ \text{end} \end{array} \tag{17}$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \ Series(\text{id}(\xi) \parallel E) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{string}; \\ \ Series(\text{id}(\xi) \parallel E) \\ \text{end} \end{array} \tag{18}$$

$$\begin{array}{l} \text{begin} \\ \text{ declare } D; \\ \ Series(E \parallel \text{id}(\xi)) \\ \text{end} \end{array} = \begin{array}{l} \text{begin} \\ \text{ declare } D, \text{id}(\xi) : \text{string}; \\ \ Series(E \parallel \text{id}(\xi)) \\ \text{end} \end{array} \tag{19}$$

Pico-recover picks up a standard identifier $\text{id}(\xi)$, that is, not tp(natural) or tp(string), with sufficient context to determine its type, and adds the corresponding declaration to the declarations section. For instance, if a value of type τ is assigned to an undeclared variable $\text{id}(\xi)$, equation (10) adds a declaration $\text{id}(\xi) : \tau$ to the program. The other equations are similar. We note that variable *Series* is overloaded. It is of type STATEMENT $\rightarrow$ SERIES as well as of type EXPR $\rightarrow$ SERIES. For more details on the lexical constructor function id, which is of type CHAR+ $\rightarrow$ ID, see [7, Sections 8.1.3 and 9].

A few details have been neglected in the specification. First, we have to exclude the values $\lambda U.t$ with U not free in t from the range of *Series* in (10)–(19). Allowing this value to be assigned to *Series* would result in extension of the declaration section with an uninstantiated declaration $\text{id}(\xi) : \tau$. The negative condition

$$(\textbf{for no } S)\ Series = \lambda U.S,$$

with S a variable of sort SERIES, takes care of this. Admittedly, this condition is not very elegant, and one would prefer it to be implicit in the notion of higher-order algebraic specification. We could have added the same condition to equation (1) to make the loop check in the implementation superfluous.

A second point is that an undeclared variable $\text{id}(\xi)$ should not be added to the declarations D more than once. This is enforced by the negative condition

$$(\textbf{for no } D_1\, \tau'\, D_2)\ \text{ declare } D; = \text{declare } D_1, \text{id}(\xi) : \tau', D_2;$$

As a result we obtain the specification

module Pico-recover
imports Pico-typecheck
variables

$Series$(STATEMENT)	$\rightarrow$ SERIES
$Series$(EXPR)	$\rightarrow$ SERIES
S	$\rightarrow$ SERIES
U	$\rightarrow$ ID
D, D_1, D_2	$\rightarrow$ {DECL , }*
E	$\rightarrow$ EXPR
ξ	$\rightarrow$ CHAR+
τ, τ'	$\rightarrow$ TYPE

equations

$$\frac{\begin{array}{rrcl} (\textbf{for no } S) & Series & = & \lambda U.S \\ (\textbf{for no } D_1\,\tau'\,D_2) & \text{declare } D; & = & \text{declare } D_1, \text{id}(\xi) : \tau', D_2; \end{array}}{\begin{array}{ll} \text{begin} & \text{begin} \\ \text{declare } D; & \text{declare } D, \text{id}(\xi) : \tau; \\ Series(\text{id}(\xi) := \text{tp}(\tau)) & = \quad Series(\text{id}(\xi) := \text{tp}(\tau)) \\ \text{end} & \text{end} \end{array}} \qquad (20)$$

$$\ldots$$

$$\ldots$$

The left-to-right rewrite rule interpretation of Pico-recover is incomplete. For instance, the program

```
begin
 declare b : string;
 while a do b := a od
end,
```

with undeclared variable a, normalizes either to

```
begin
 declare b : string, a : natural;
 tp(string) := tp(natural)
end,
```

or to

```
begin
 declare b : string,  a : string;
 while tp(string) do od
end.
```

If an undeclared variable is used inconsistently, the implementation adds one of the possible declarations for it. Hence, the incompleteness of the rewrite rule interpretation reflects the non-deterministic character of the recovery process, and the specification is not a suitable candidate for (higher-order) critical-pair completion (cf. Hussmann [13]).

Pico-recover actually performs *higher-order narrowing* (Section 1.1). Let N be a Pico-typecheck normal form and let N' be obtained from N by adding an uninstantiated declaration $\mathrm{id}(\xi) : \tau$ at the end of its declarations section. Narrowing of N' with equation (1) followed by narrowing with equations (2)–(9) instantiates ξ and τ to values Pico-recover would have obtained by rewriting N with equations (10)–(19). Hence, an implementation of higher-order algebraic specifications based on narrowing rather than rewriting would, at least in principle, make Pico-recover largely superfluous.

3 Adding block structure to Pico

3.1 Higher-order abstract syntax

We extend Pico to Pico-B by adding block structure to it. The obvious extension of Pico-syntax to

module FO-Pico-B-syntax
imports Pico-syntax (Section 2.1)
functions block DECLS SERIES end → STATEMENT

yields a suitable concrete syntax, but the corresponding abstract syntax is first-order and does not exploit the possibilities of a typed λ-term representation as discussed by Huet and Lang [11] and Pfenning and Elliot [23]. Furthermore, it does not allow us to reuse Pico-typecheck and Pico-recover.

If local Pico-B variables are represented by λ-bound variables, higher-order matching automatically takes the block structure into account. As a consequence, Pico-typecheck and Pico-recover can be reused in a straight-

forward way. From this perspective, the block

block
declare a_1 : type$_1$, ..., a_n : type$_n$;
...
end.

corresponds to the second-order term

$$\text{block}(\lambda a_1 \ldots a_n.\ \text{pair}(\text{declare}(\text{decl}(a_1, \text{type}_1), \ldots, \text{decl}(a_n, \text{type}_n)), \cdots)),$$

rather than to the first-order term

$$\text{block}(\text{declare}(\text{decl}(a_1, \text{type}_1), \ldots, \text{decl}(a_n, \text{type}_n)), \cdots)$$

defined by FO-Pico-B-syntax.

An extension of Pico-syntax that (partly) defines such a higher-order representation is:

module	HO-Pico-B-syntax	
imports	Pico-syntax	(Section 2.1)
functions	block ID* → PAIR end	→ STATEMENT
	DECLS SERIES	→ PAIR

According to HO-Pico-B-syntax, the "block ··· end" constructor is of type (ID* → PAIR) → STATEMENT. No concrete syntax is associated with the binder. HO-Pico-B-syntax is not complete. It does not say that the identifiers occurring in the DECLS-part should become bound variables in the PAIR-part. We are still developing a suitable notation for this as part of the extension of SDF to higher-order abstract syntax [9]. See also the recent proposal by Felty [6].

3.2 Basic static semantics

Apart from an additional equation, the basic static semantics specification of Pico-B can be based on Pico-typecheck:

module	Pico-B-typecheck	
imports	HO-Pico-B-syntax,	(Section 3.1)
	Pico-typecheck	(Section 2.2)
variables	*Series*(ID)	→ SERIES
	D_1, D_2	→ {DECL , }*
	X	→ ID
	τ	→ TYPE

equations

$$\begin{array}{l}\text{declare } D_1, X : \tau, D_2; \\ Series(X)\end{array} \quad = \quad \begin{array}{l}\text{declare } D_1, X : \tau, D_2; \\ Series(\text{tp}(\tau))\end{array} \tag{21}$$

Equation (21) is identical to (1), except that it applies to the pair-constructor of sort PAIR (which has no concrete representation) rather than the "begin $\cdots$ end" constructor of sort PROGRAM. Because of the higher-order abstract syntax enforced by the "block $\cdots$ end" constructor, the application of (21), unlike that of (1), requires the abstraction rule

$$\frac{\vdash t_1 = t_2}{\vdash \lambda X.t_1 = \lambda X.t_2}.$$

This rule of higher-order equational logic corresponds to reduction under abstraction in the rewrite rule interpretation (see [8, Section 2.3]).

3.3 Recovery from undeclared variables

Recovery from undeclared variables is defined in the same way as before. Pico-recover is still valid, and can be reused:

module	Pico-B-recover	
imports	Pico-B-typecheck,	(Section 3.2)
	Pico-recover	(Section 2.3)

Declarations for undeclared variables are added to the outermost declarations section of the program.

References

[1] R. Bahlke and G. Snelting, The PSG system: from formal language definitions to interactive programming environments, *ACM Transactions on Programming Languages and Systems*, **8** (1986) 547–576.

[2] J.A. Bergstra, J. Heering, and P. Klint, eds., *Algebraic Specification* (ACM Press/Addison-Wesley, 1989).

[3] A. van Deursen, P. Klint, and F. Tip, Origin tracking, Report CS-R9230, CWI, Amsterdam, July 1992.

[4] T.B. Dinesh and F. Tip, Animators and error reporters for generated programming environments, Report CS-R9253, CWI, Amsterdam, December 1992.

[5] G. Dowek, A second-order pattern matching algorithm for the cube of typed λ-calculi, in: A. Tarlecki, ed., *Mathematical Foundations of Computer Science 1991*, Lecture Notes in Computer Science, Vol. 520 (Springer-Verlag, 1991) 151–160.

[6] A. Felty, Defining object-level parsers in λProlog (extended abstract), in: D. Miller, ed., *Proceedings of the 1992 Workshop on the λProlog Programming Language*, Report MS-CIS-92-86, University of Pennsylvania, Philadelphia, 1992, 87–99.

[7] J. Heering, P.R.H. Hendriks, P. Klint, J. Rekers, The syntax definition formalism SDF—reference manual, *SIGPLAN Notices*, **24**(11) (1989) 43–75.

[8] J. Heering, Implementing higher-order algebraic specifications, in: D. Miller, ed., *Proceedings of the 1992 Workshop on the λProlog Programming Language*, Report MS-CIS-92-86, University of Pennsylvania, Philadelphia, 1992, 141–157.

[9] J. Heering, Extension of the syntax definition formalism SDF to higher-order abstract syntax, CWI, Amsterdam, in preparation.

[10] P.R.H. Hendriks, *Implementation of Modular Algebraic Specifications*, Ph.D. Thesis, University of Amsterdam, 1991.

[11] G. Huet and B. Lang, Proving and applying program transformations expressed with second-order patterns, *Acta Informatica*, **11**(1978) 31–55.

[12] H. Hussmann, Rapid prototyping for algebraic specifications—RAP system user's manual, Report MIP-8504, Universität Passau, 2nd extended edition, 1987.

[13] H. Hussmann, Non-deterministic algebraic specifications, Report TUM-I9104, Technische Universität München, 1991.

[14] J.-P. Jouannaud and M. Okada, A computation model for executable higher-order algebraic specification languages, in: *Proceedings of the*

Sixth Annual IEEE Symposium on Logic in Computer Science (IEEE Computer Society Press, 1991) 350–361.

[15] P. Klint, A meta-environment for generating programming environments, Report CS-R9064, CWI, Amsterdam, 1990. To appear in *ACM Transactions on Software Engineering Methodology.*

[16] K. Meinke, Universal algebra in higher types, *Theoretical Computer Science*, **100** (1992) 385–417.

[17] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, in: P. Schroeder-Heister, ed., *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 475 (Springer-Verlag, 1991) 253–281.

[18] B. Möller, Algebraic specification with higher-order operators, in: L.G.L.T. Meertens, ed., *Program Specification and Transformation* (North-Holland/IFIP, 1987) 367–398.

[19] G. Nadathur and D. Miller, An overview of λProlog, in: R.A. Kowalsi and K.A. Bowen, eds., *Logic Programming—Proceedings of the Fifth International Conference and Symposium*, Vol. 1 (The MIT Press, 1988) 810–827.

[20] T. Nipkow, Higher-order critical pairs, in: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1991) 342–349.

[21] L.C. Paulson, Isabelle: The next 700 theorem provers, in: P. Odifreddi, ed., *Logic and Computer Science* (Academic Press, 1990) 361–385.

[22] K. Parsaye-Ghomi, Higher-order abstract data types, Report CSD-820112, Computer Science Department, University of California, Los Angeles, January 1982.

[23] F. Pfenning and C. Elliott, Higher-order abstract syntax, *SIGPLAN Notices*, **23**(7) (1988) 199–208.

[24] A. Poigné, On specifications, theories, and models with higher types, *Information & Control*, **68** (1986) 1–46.

[25] W. Snyder, *A Proof Theory for General Unification*, Birkhäuser, 1991.

[26] H.R. Walters, The static semantics of POOL, in: [2, Chapter 4].