Basic Research in Computer Science

BRICS NS-00-6 Mosses & de Moura (eds.): AS 2000 Proceedings

Proceedings of the Third International Workshop on

Action Semantics
AS 2000

Recife, Brazil, May 15–16, 2000

Peter D. Mosses
Hermano Perrelli de Moura
(editors)

BRICS Notes Series **NS-00-6**

ISSN 0909-3206 **August 2000**

Copyright © 2000, Peter D. Mosses & Hermano Perrelli de Moura (editors).
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/00/6/`

AS 2000

Third International Workshop on

Action Semantics

15-16 May 2000, Recife, Brazil

Proceedings

Foreword

Action Semantics[1] is a practical framework for formal semantic description of programming languages. Since its appearance in 1992, Action Semantics has been used to describe major languages such as Pascal, SML, ANDF, and Java, and various tools for processing action-semantic descriptions have been developed.

The AS 2000 workshop included reports of recent achievements with the foundations and applications of Action Semantics, presentations and demonstrations of tool support for action-semantic descriptions, and discussion of a proposal for a new (and significantly simpler) version of Action Notation.

AS 2000 was, like the previous workshops in this series [**?**,**?**], intended primarily for those working with Action Semantics and related approaches, but participation was open to all those familiar with the basic ideas of Action Semantics. The 2-day workshop allowed 45-minute presentations of the submitted work, with reasonable time for discussion. Holding the workshop as a satellite event of SBLP'2000 (the 4th Annual Brazilian Symposium on Programming Languages) in Recife not only gave participants the opportunity to combine attendance of the two events, but also drew extra attention to Action Semantics.

This was the first time that the AS workshop took place outside Europe, and the extent of participation by groups working on Action Semantics in Brazil was particularly gratifying: apart from the group in Recife, there were reports on current work with Action Semantics from three separate projects at PUC in Rio de Janeiro, as well as from one at UFPR in Curitiba. Special thanks to the invited speaker, David Watt (presenting joint work with Deryck Brown), and to all those who gave presentations at AS 2000.

In 2001, LDTA, a 1-day workshop on Language Descriptions, Tools, and Applications, is to be held as a satellite event of ETAPS, in Genova, Italy. Action Semantics is mentioned as one of the topics of interest of LDTA in the Call for Papers[2], and authors are encouraged to submit full papers by the end of November 2000. Consequently, no separate AS workshop is planned for 2001, and further discussion of the proposed new version of Action Notation will have to take place on the AS mailing list: `action-semantics@brics.dk`. Proposals for hosting an AS workshop in 2002 are most welcome, and should be sent to Peter Mosses by March 2001.

Peter D. Mosses
BRICS & Dept. of Computer Science
Univ. of Aarhus, Denmark

Hermano Perrelli de Moura
Centre for Informatics, UFPE
Recife, Brazil

[1] http://www.brics.dk/Projects/AS/
[2] http://www-sop.inria.fr/oasis/LDTA/ldta.html

Final Programme

Monday 15 May:

09:00 **Theoretical Foundations**

Action Semantics for Logic Programming Languages
Luis Carlos de Souza Meneses, Hermano Perrelli de Moura, Geber Lisboa Ramalho (UFPE, Recife, Brazil)

09:45 *CASL and Action Semantics*
Peter D. Mosses (BRICS & University of Aarhus, Denmark)

10:30 Coffee

11:00 **Theoretical Foundations, ctd.**
Postfix Transformations for Action Notation
Kent Lee (Luther College, Iowa, USA)

11:45 *Modular SOS and Action Semantics*
Peter D. Mosses (BRICS & University of Aarhus, Denmark)

12:30 Lunch

14:00 **Invited talk:**

Formalizing the Semantics of Java
David A. Watt (University of Glasgow, Scotland), Deryck F. Brown (The Robert Gordon University, Aberdeen, Scotland)

15:15 **Recent Action-Semantic Descriptions**

A Formal Description of SNMPv3 Standard Applications using Action Semantics
Diógenes Cogo Furlan, Martin A. Musicante, Elias Procópio Duarte Jr. (UFPR, Curitiba, Brazil)

16:00 Coffee

16:30 **Recent Action-Semantic Descriptions, ctd.**
D2L: A Design Description Language
Sergio E.R. de Carvalho†, Christina von Flach G. Chavez, Sylvia de Oliveira e Cruz (PUC, Rio de Janeiro, Brazil)

17:15 *An Action Semantics for STG*
Francisco Heron de Carvalho Junior, Hermano Perrelli de Moura, Ricardo Massa Ferreira Lima, Rafael Dueire Lins (UFPE, Recife, Brazil)

18:00 Close of Session

21:00 **Workshop Dinner**

Tuesday 16 May:

09:00 **Tools and Demos**

Abaco System : An Action Semantics Based Compiler Generation System
Luis Carlos de Souza Meneses, Hermano Perrelli de Moura (UFPE, Recife, Brazil)

09:45 *Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic*
Christiano Braga, E. Hermann Haeusler (PUC, Rio de Janeiro, Brazil), José Meseguer (SRI International, USA), Peter D. Mosses (BRICS & University of Aarhus, Denmark)

10:30 Coffee

11:00 **Tools and Demos, ctd.**

A Modular Implementation of Action Notation
Leonardo de Moura, Carlos de Lucena, E. Hermann Haeusler (PUC, Rio de Janeiro, Brazil)

11:45 Discussion

12:30 Lunch

14:00 **AN-2: The Proposed New Version of Action Notation**

Introduction to AN-2
Søren B. Lassen (Digital Fountain, USA), Peter D. Mosses (BRICS & University of Aarhus, Denmark), David A. Watt (University of Glasgow, Scotland)

15:00 **Discussion**

16:00 Coffee

16:30 **Coordination of Projects**

18:00 End of Workshop

Sponsorship

The organizers of AS 2000 gratefully acknowledge funding and sponsorship from:

BRICS (Centre for Basic Research in Computer Science, Denmark)[3]

Centre for Informatics, UFPE, Recife, Brazil

[3] Established by the Danish National Research Foundation, in collaboration with the Universities of Aarhus and Ålborg.

Table of Contents

Invited Talk

Formalising the Dynamic Semantics of Java 1

David A. Watt, Deryck F. Brown

The Proposed New Version of Action Notation

An Introduction to AN-2: The Proposed New Version of Action Notation . 19

Søren B. Lassen, Peter D. Mosses, and David A. Watt

Theoretical Foundations

Postfix Transformations for Action Notation 37

Kent D. Lee

Action Semantics for Logic Programming Languages 47

Luis Carlos Menezes, Hermano Perrelli de Moura, Geber Ramalho

CASL and Action Semantics 62

Peter D. Mosses

Modular SOS and Action Semantics (Abstract) 79

Peter D. Mosses

Recent Action-Semantic Descriptions

An Action Semantics for the D2L Design Description Language 81

Christina von Flach G. Chavez, Sylvia de Oliveira e Cruz, Sergio E.R. de Carvalho

An Action Semantics for STG 98

Francisco Heron de Carvalho Junior, Hermano Perrelli de Moura, Ricardo Massa Ferreira Lima, Rafael Dueire Lins

A Formal Description of SNMPv3 Standard Applications using Action Semantics 118

Diógenes Cogo Furlan, Martín A. Musicante, Elias Procópio Duarte Jr.

Tools and Demonstrations

Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic (Abstract) 133

Christiano de O. Braga, E. Hermann Haeusler, José Meseguer, Peter D. Mosses

A Modular Implementation of Action Notation 134

Leonardo M. de Moura, Carlos J. P. de Lucena and E. Hermann Haeusler

Author Index

Braga, Christiano de O., 133
Brown, Deryck F., 1

Carvalho Junior, Francisco Heron de, 98
Carvalho, Sergio E.R. de, 81
Chavez, Christina von Flach G., 81
Cruz, Sylvia O., 81

Duarte Jr., Elias Procópio, 118

Furlan, Diógenes Cogo, 118

Haeusler, E. Hermann, 133, 134

Lassen, Søren B., 19
Lee, Kent D., 37
Lima, Ricardo Massa Ferreira , 98
Lins, Rafael Dueire, 98
Lucena, Carlos J. P. de , 134

Menezes, Luis Carlos, 47
Meseguer, José, 133
Mosses, Peter D., 19, 62, 79, 133
Moura, Hermano Perrelli de, 47, 98
Moura, Leonardo M. de, 134
Musicante, Martin A., 118

Ramalho, Geber, 47

Watt, David A., 1, 19

Formalising the Dynamic Semantics of Java*

David A. Watt[1] and Deryck F. Brown[2]

[1] Department of Computing Science, University of Glasgow,
Glasgow G12 8QQ, Scotland. Email: daw@dcs.gla.ac.uk
[2] School of Computer and Math Sciences, The Robert Gordon University,
St Andrew Street, Aberdeen AB25 1HG, Scotland. Email: db@scms.rgu.ac.uk

Abstract. In this paper, we review three different formal descriptions of the dynamic semantics of the Java programming language. The first description uses denotational semantics [2], the second uses abstract state machines [3], and the third uses action semantics [4].
We compare these descriptions by systematically comparing how they deal with selected Java language constructs: control flow using while and break statements; methods, invocation expressions, and return statements; and exceptions using throw and try-catch statements. From these comparisons we draw conclusions about the success or otherwise of the three reviewed descriptions, and about the underlying formalisms themselves.

1 Introduction

In this paper, we study three different formal descriptions of the Java programming language, each of which uses a different semantic formalism. The descriptions to be studied are:

- Alves-Foss and Lam's denotational-semantic (DS) description [2]
- Börger and Schulte's abstract state machines (ASM) description [3]
- our own action-semantic (AS) description [4].

In each case, we give an overview of the overall structure of the description, and show in detail the semantics of selected Java language constructs: control flow using while and break statements; method invocation and the return statement; and exceptions using the throw and try-catch statements. To facilitate comparison, we have slightly modified some of the notation used in the original papers.

Only the ASM description claims to describe the entire Java language. The DS description omits threads, and the AS description omits both threads and overloading.

The remainder of this paper considers the DS description in Section 2; the ASM description in Section 3; and the AS description in Section 4. Section 5 concludes by comparing the three descriptions for correctness, intelligibility, modularity, and extensibility.

* Java is a trademark of Sun Microsystems Inc.

2 Denotational Semantics

In this section we study Alves-Foss and Lam's DS description of Java's semantics [2].

[2] uses Greek letters for names of variables and semantic functions. For the benefit of our readers, the Greek letters are replaced by more meaningful identifiers in the extracts presented here. Also, [2] uses universal quantification to bind variables; here the more conventional λ-notation is used.

The DS description is written in the continuation passing style. Using this style, the semantics of each syntactic construct is represented by a higher-order function, where the type of this function depends on the syntactic construct being specified. One of the arguments of this function is a *continuation*, which is a function used to represent the behaviour of the remainder of the program.

As in most classical DS descriptions, there is an *environment* that provides the bindings of identifiers declared in the program, and a *store* that maps storage locations to their contents. In this particular DS description, the environment is also used to hold auxiliary semantic variables, which are used to propagate information about the current declaration, the flow of control, and the current object.

Several different kinds of continuation are used in the description: package, declaration, statement, expression, and location continuations. In our extracts, we will only need three of these: expression, statement, and declaration continuations. Their types are as follows:

$$ExprCont = Value \times Type \times Store \to Answer$$
$$StmtCont = Env \times Store \to Answer$$
$$DeclCont = Env \to Env$$

and they are typically used as follows:

$$econt(val, typ, sto) = ans$$
$$scont(env, sto) = ans$$
$$dcont(env_1) = env_2$$

- An expression continuation, *econt*, represents the remainder of the program following the evaluation of an expression. It receives a value, *val*, a type, *typ*, and a store, *sto*, where typically *val* and *typ* are the result of evaluating the expression, and *sto* is the resulting store.
- A statement continuation, *scont*, represents the remainder of the program following the execution of a statement. It receives an environment, *env*, and a store, *sto*, that are typically the modified environment and store produced by executing a single statement.
- A declaration continuation, *dcont*, represents the declarations produced by the remainder of the program. It receives an environment, env_1, that is typically the environment produced by elaborating a single declaration.

For each syntactic construct, there is a corresponding semantic function: *eval* for expressions, *exec* for statements, and *elab* for declarations. Their types are as follows:

$$
\begin{aligned}
&\mathit{eval} :: [\![\ \mathit{Expr}\ ]\!] \to \mathit{Env} \to \mathit{ExprCont} \to \mathit{Store} \to \mathit{Answer}\\
&\mathit{exec} :: [\![\ \mathit{Stmt}\ ]\!] \to \mathit{Env} \to \mathit{StmtCont} \to \mathit{Store} \to \mathit{Answer}\\
&\mathit{elab} :: [\![\ \mathit{Decl}\ ]\!] \to \mathit{Env} \to \mathit{DeclCont} \to \mathit{Env}
\end{aligned}
$$

The effect of each of these semantic functions is as follows:

- An expression, *Expr*, is evaluated using the semantic function *eval*. This function takes an initial environment, *env*, an expression continuation, *econt*, and an initial store, *sto*. The expression *Expr* is evaluated, using *env* and *sto*, to produce a value, *val*, its type, *typ*, and a modified store sto_1. If the expression evaluation terminates normally, the triple (val, typ, sto_1) is simply passed to *econt*, and program execution continues. If the expression throws an exception, *econt* is ignored, and program execution continues using some other continuation, typically fetched from an environment.
- A statement, *Stmt*, is executed using the semantic function *exec*. This function takes an initial environment, *env*, a statement continuation, *scont*, and an initial store, *sto*. The statement *Stmt* is executed, using *env* and *sto*, to produce a modified environment, env_1, and a modified store, sto_1. If the statement terminates normally, env_1 and sto_1 are passed to *scont*, and program execution continues. If the statement throws an exception, *scont* is ignored, and program execution continues using some other continuation, typically fetched from the environment.
- A declaration, *Decl*, is elaborated using the semantic function *elab*. This function takes an initial environment, *env*, and a declaration continuation, *dcont*. The declaration *Decl* is elaborated to produce a modified environment, env_1, which is passed to *dcont*, and the elaboration continues. In Java, the elaboration of a declaration cannot terminate exceptionally.

2.1 Control flow

Let us now study the description of Java's control structures, exemplified by while and (unlabeled) break statements. In Java, executing an unlabeled break has the effect of jumping to the statement after the current loop or switch statement.

The semantic equation for the while statement is as follows:

(1)
$$
\begin{aligned}
&\mathit{exec}\ [\![\ \texttt{while (} \mathit{Expr} \texttt{)}\ \mathit{Stmt}\ ]\!]\ \mathit{env}\ \mathit{scont}\ \mathit{sto} =\\
&\quad \mathit{scont}_1(\mathit{env}[\&\mathit{break} \leftarrow \mathit{scont}], \mathit{sto}) \textbf{ where rec}\\
&\quad\quad \mathit{scont}_1 = \lambda(\mathit{env}_1, \mathit{sto}_1).\ \mathit{eval}\ [\![\ \mathit{Expr}\ ]\!]\ \mathit{env}_1\ \mathit{econt}\ \mathit{sto} \textbf{ where}\\
&\quad\quad\quad \mathit{econt} = \lambda(\mathit{val}, \mathit{typ}, \mathit{sto}_2).\\
&\quad\quad\quad\quad \textbf{if } \mathit{val} = \mathit{true}\\
&\quad\quad\quad\quad \textbf{then } \mathit{exec}\ [\![\ \mathit{Stmt}\ ]\!]\ \mathit{env}_1\ \mathit{scont}_1\ \mathit{sto}_2\\
&\quad\quad\quad\quad \textbf{else } \mathit{scont}(\mathit{env}, \mathit{sto}_2)
\end{aligned}
$$

This is defined recursively using the continuations $scont_1$ (which starts with the evaluation of *Expr*) and *econt* (which either starts with the execution of *Stmt* or is the rest of the program). The loop body, *Stmt*, is executed in an

environment where the auxiliary variable *&break* is bound to *scont*, the continuation representing the rest of the program. This variable is used by the break statement.

The semantic equation for the unlabeled break statement is as follows:

(2) $$exec \; [\![\texttt{break ;}]\!] \; env \; scont \; sto = \\ scont_1(env, sto) \textbf{ where} \\ scont_1 = env.getStmtCont(\&break)$$

This is relatively straightforward. Control is passed to the statement continuation, $scont_1$, bound to *&break*. This binding is created by the enclosing construct, e.g., equation (1). Note that *scont* is ignored, being the continuation representing normal control flow through the enclosing construct.

2.2 Methods

Let us now study the description of methods. Note that [2] appears not to handle instance method invocation, where the target object is determined by an expression, but only class method invocation.

A method is represented by the partial application of *exec* to the method body. As such, this is a function that takes an environment, a statement continuation, and a store, and returns an answer.

The semantic equation for the class method invocation (which is an expression in Java) is as follows:

(3) $$eval \; [\![\; Name \; \texttt{(} \; Arglist \; \texttt{)} \;]\!] \; env \; econt \; sto = \\ eval \; [\![\; Arglist \;]\!] \; env \; econt_1 \; sto \textbf{ where} \\ econt_1 = \lambda(vals, typs, sto_1).\; meth(env, scont, sto_1) \textbf{ where} \\ sig = getSigs(vals) \textbf{ and} \\ meth = env.getMethod(\textbf{fst } (id[\![Name]\!] \; env), sig) \textbf{ and} \\ scont = \lambda(env_2, sto_2). \\ econt(env_2[\&returnVal], env_2[\&returnType], sto_2)$$

This involves evaluating the list of arguments, *Arglist*, using the expression continuation representing the method body, $econt_1$. Identifying the method to invoke involves constructing a method signature, *sig*, using the values of the actual parameters, *vals*. Overloaded method names are handled by the auxiliary function *getMethod* by using the method signature *sig*. From [2], it is unclear how the actual parameters, *vals*, are passed to the method, *meth*. The only apparent route is via the operation of the auxiliary function *getSigs*, but the behaviour of this operation is not given. The curious term "**fst**$(id[\![Name]\!]env)$" is used (amongst other things) to convert the syntactic name *Name* into its corresponding semantic value, which is a pair consisting of the name itself and its type. The method body, *meth*, is passed its environment when invoked, which is unusual for a statically-bound language. This environment is required, however, to access the (dynamic) values bound to auxiliary semantic variables. The result of the method is stored in the environment using two semantic variables,

$\&returnVal$ and $\&returnType$. These values are retrieved by *scont* and passed to *econt* along with the store produced by the method body, sto_2.

The semantic equation for the return statement is as follows:

(4) $$\begin{array}{l} exec\ [\![\ \texttt{return}\ Expr\ \texttt{;}\ ]\!]\ env\ scont\ sto = \\ \quad eval\ [\![\ Expr\ ]\!]\ env\ econt\ sto\ \textbf{where} \\ \quad\quad econt = \lambda(val, typ, sto_1).\ scont_1(env_1, sto_1)\ \textbf{where} \\ \quad\quad\quad scont_1 = env.getStmtCont(\&return)\ \textbf{and} \\ \quad\quad\quad typ_1 = env[\&returnType]\ \textbf{and} \\ \quad\quad\quad val_1 = promote(typ_1, (val, typ))\ \textbf{and} \\ \quad\quad\quad env_1 = env[\&returnVal \leftarrow val_1] \end{array}$$

Since equation (3) expects to find the result in the environment, the return statement must evaluate the return expression and construct a new environment, env_1, with the required binding. Note that equations (3) and (4) both fetch the expected return type, $\&returnType$, from the environment. It is unclear from [2] how this return type is placed in the environment received by the method body. Equation (4) also ignores *scont*, which represents the rest of the method body, and instead transfers control to $scont_1$, which is bound to the auxiliary variable $\&return$. This will immediately pass control back to the location of the method invocation, without executing the rest of the method body.

2.3 Exceptions

Let us now study the description of exceptions. We will consider the throw and try-catch statements.

The semantic equation for the throw statement is as follows:

(5) $$\begin{array}{l} exec\ [\![\ \texttt{throw}\ Expr\ \texttt{;}\ ]\!]\ env\ scont\ sto = \\ \quad eval\ [\![\ Expr\ ]\!]\ env\ econt\ sto\ \textbf{where} \\ \quad\quad econt = \lambda(val, typ, sto_1).\ scont_1(env_2, sto_1)\ \textbf{where} \\ \quad\quad\quad env_2 = env_1[\&thrown \leftarrow (val, typ)]\ \textbf{and} \\ \quad\quad\quad scont_1 = env[\&throw] \end{array}$$

The throw statement transfers control to $scont_1$, which is bound to the auxiliary variable $\&throw$ in *env*. It passes to $scont_1$ a modified environment, env_2, where the auxiliary variable $\&thrown$ is bound to the result of evaluating *Expr*. Equation (5) is incorrect when this result is null. In this case, the throw statement should throw a `NullPointerException`, and not the value null as specified in the equation.

The semantic equation for the try-catch statement is as follows:

(6) $exec \llbracket \texttt{try}\ Block\ Catches \rrbracket\ env\ scont\ sto =$
$exec \llbracket\ Block \rrbracket\ env_1\ scont_1\ sto$ **where**
$env_1 = env[\&throw \leftarrow scont_2]$ **and**
$scont_1 = \lambda(env_2, sto_2).\ scont(env, sto_2)$ **and**
$scont_2 = \lambda(env_2, sto_2).$
$exec \llbracket\ Catches \rrbracket\ env\ scont_3\ sto_2$ **where**
$scont_3 = \lambda(env_3, sto_3).$
if $(env_3[\&thrown] = (null,$ "V"$))$
then $scont(env_3, sto_3)$
else $(env_3[\&throw])(env_3, sto_3)$

The try-catch statement performs *Block* in an environment, env_1, where the auxiliary variable $\&throw$ is bound to $scont_2$, which represents the catch-clauses, *Catches*. It is vital that $scont_2$ ignores the environment, env_2, passed to it, and instead uses the original environment, *env*. env_2 contains the current binding for $\&throw$, which represents the catch-clauses associated with this try-catch statement, whereas *env* contains a binding for $\&throw$ that represents some outer statement. If no exception is thrown by *Block*, control is passed to $scont_1$, which discards the environment it is passed, effectively deactivating the exception handlers in the catch-clauses, and replaces it with the original environment, *env*. Equation (6) ignores the situation where the value thrown is null, since the description uses the null value to signal that an exception has been handled. In Java, the value of an exception can never be null. However the error in equation (5) has the consequence that throwing the value null will be ignored altogether.

When an exception is handled, the catch-clauses must be inspected in turn until one is found that matches the class of the thrown value. If no match is found, the exception is re-thrown to be handled at an outer level. The equation for a single catch-clause is as follows:

(7) $exec \llbracket \texttt{catch (}\ Formal\ \texttt{)}\ Block \rrbracket\ env\ scont\ sto =$
let $(exc, typ) = env[\&thrown]$ **and**
$(id, typ_1) = id \llbracket\ Formal \rrbracket\ env$
in if $typ = typ_1$
then $exec \llbracket\ Block \rrbracket\ env_1\ scont\ sto_1$ **where**
$env_1 = env[\&thrown \leftarrow (null,$ "V"$)]$ **and**
$sto_1 = sto[env[id] \leftarrow exc]$
else $scont(env, sto)$

This first determines the value and type of the thrown value, (exc, typ), and then compares *typ* with the type of exception handled by this catch-clause, typ_1. In Java, if the type of the handler is assignment compatible with the type of the thrown value, the handler should be performed. However, equation (7) incorrectly specifies that these types should be equal. When a handler is chosen, the block it contains should be executed in an environment where the formal parameter, *id*, is bound to a location initialised with the exception value, *exc*. Although equation (7) constructs a store, sto_1, containing the exception value, it does not construct an environment containing the required binding.

2.4 Summary

In our view, this DS description of Java is highly unpromising and any attempt to develop it into a complete description of Java is likely to fail. Our main observations are as follows:

- The DS description contains numerous errors, both typographic and conceptual. Apart from the errors mentioned above, there are many other flaws. For example, the for statement is specified by a translation to the corresponding while statement. However, this translation produces an incorrect scope for any variables declared in the initialisation expression of the for statement, which should be restricted to the body of the for statement and not the block that contains it.
- The DS description is very difficult to read, even after renaming variables and semantic functions as in the above extracts. The continuation passing style relies heavily on the reader keeping track of the different continuations used and the arguments they are given. This is particularly true when so much dynamic information is hidden in the environment.
- Like most classical descriptions, this DS description lacks any modular structure and would be difficult to modify, since the semantic equations are tightly coupled to the choice of semantic domains.
- The DS description is not entirely compositional.
- The DS description misuses the environment to store auxiliary semantic variables. A clean separation of the dynamic part of the environment (containing the auxiliary information), and the static part (containing the program identifiers) would be preferable and easier to understand.
- The DS description cannot be extended to deal with threads. To specify the concurrent aspects of Java, a completely new DS description would have to be written using power domains.

3 Abstract State Machines

In this section we study Börger and Schulte's ASM description of Java's semantics [3].

The description is structured as a series of sub-descriptions, one for each of five nested sub-languages:

- Java$_{\mathcal{I}}$ contains expressions, statements and blocks.
- Java$_{\mathcal{C}}$ adds classes and class methods. This is an imperative language with simple encapsulation.
- Java$_{\mathcal{O}}$ adds objects and instance methods. This is a true object-oriented language.
- Java$_{\mathcal{E}}$ adds exceptions. This corresponds to the sequential subset of Java.
- Java$_{\mathcal{T}}$ adds threads. This is essentially the full language.

The ASM description assumes a highly simplified abstract syntax. For example, it assumes that for and do statements have been reduced to equivalent while

statements, and switch statements to equivalent if statements. Also eliminated are compound assignment operators (such as `+=` and `-=`), prefix and postfix operators (`++` and `--`), conditional operators (`&&` and `||`), variable initialisers, arrays and strings.

The ASM description also invents abstract syntax not present in Java itself. For example, a method invocation expression is extended with information about the kind and signature of the invoked method.

The ASM description, as its name suggests, works essentially by defining a state space together with transitions between states. The state has a complex structure, part of which is explained here.

The most important part of the state is the *stack*, more precisely a triple of stacks:

$$stack \equiv (taskStack, valStack, locStack)$$

$$taskStack : Phrase^*$$
$$valStack : (Exp \to Value)^*$$
$$locStack : (Var \to Value)^*$$

$$task \equiv top(taskStack)$$
$$val \equiv top(valStack)$$
$$loc \equiv top(locStack)$$

Each element of *taskStack* is a so-called *task*, i.e., a phrase about to be executed. A task may be understood as an abstract program counter. The topmost task in the stack, *task*, is the current program counter, and the underlying tasks are return addresses.

Each element of *locStack* contains the current values of some method's local variables. Thus *locStack* may be understood as a stack of frames, where the topmost frame, *loc*, belongs to the currently-active method.

Each element of *valStack* contains the value of each expression in some method. We can think of each expression's value being assigned to an anonymous temporary variable, and *valStack* is a stack of such temporaries.

The term *abruption* is coined to mean an abrupt termination of a phrase: a break from a loop, a return from a method, or a throw of an exception. When an abruption occurs, the reason for it is recorded in the following state variable:

$$mode : Reason$$

In Java$_{\mathcal{O}}$, the state also contains:

$$classOf : Reference \to Class$$
$$dyn : Reference \times FieldSpec \to Value$$

Let ref be a reference to an object. Then the object to which r refers contains a class tag, $classOf(ref)$, and a group of field values, $dyn(ref, field)$.

Transitions between states are captured by the following functions:

$$fst : Phrase \to Phrase$$

$nxt : Phrase \rightarrow Phrase$
$up : Phrase \rightarrow Phrase$

Let $task$ be a phrase (statement or expression). Then $fst(task)$ is the first child phrase of $task$ to be executed; $nxt(task)$ is the next sibling phrase to be executed when $task$ terminates normally, or $task$'s parent phrase if $task$ has no such sibling; and $up(task)$ is the next (usually ancestor) phrase that might handle an abruption in $task$, or **finished** if $task$ has no such ancestor. The ASM description defines fst, nxt, and up appropriately for each phrase in the language.

3.1 Control flow

Let us now study the description of Java's control structures, exemplified by while, break, and labeled statements.

For the while statement, fst and nxt are defined as follows:

(1) **let** $stm = (\texttt{while (}\ exp\ \texttt{)}\ stm_1)$ **in**
$fst(stm) = fst(exp)$
$nxt(exp) = stm$
$nxt(stm_1) = fst(exp)$

Written in terms of these, the semantic rule for a while statement is as follows:

(2) **if** $task$ is $(\texttt{while (}\ exp\ \texttt{)}\ stm_1)$ **then**
if $val(exp) = true$ **then**
$task := fst(stm_1)$
else
$task := nxt(task)$

The semantic rules for break and labeled statements are as follows:

(3) **if** $task$ is $(\texttt{break}\ lab\ \texttt{;})$ **then**
$mode := Break(lab)$
$task := up(task)$

(4) **if** $task$ is $(lab\ \texttt{:}\ stm)$ **then**
if $mode = Break(lab)$ **then**
$mode := undef$
$task := nxt(task)$
else
$task := up(task)$

If stm abrupts by executing "`break` lab `;`", then control flows on to the corresponding labeled statement's next sibling, $nxt(task)$. Otherwise control flows up to an enclosing labeled statement, $up(task)$.

3.2 Methods

Let us now study the description of instance methods (in Java$_{\mathcal{O}}$).

The syntax of an instance method invocation is assumed to have been extended with the sub-phrase "{*kind*}", where *kind* has three possible values:

- *Nonvirtual*: the invoked method is private, and must be in the current class.
- *Virtual*: the invoked method is to be selected dynamically, starting with the target object's class.
- *Super*: the invoked method is to be selected dynamically, starting with the current class's superclass.

Moreover, the method identifier is assumed to have been replaced by *methodSpec*, which includes signature information necessary for overload resolution. Here we see a particularly clear example of invented abstract syntax, which simplifies the semantic rule but glosses over how to specify the mapping from concrete syntax to abstract syntax.

The following is the semantic rule for an instance method invocation expression:

(5) **if** $task$ is (exp `.` $methodSpec$ {$kind$} `(` $exp_1, \ldots, exp_n$ `)`)
$\quad\quad\quad \wedge\ val(exp) \neq null$ **then**
$\quad stack := invoke(\langle val(exp)\rangle vals, \langle$`this`$\rangle vars, fst(body), stack)$
where
$\quad (vars, body) = instMethod(methodSpec, class, kind)$
$\quad vals = \langle val(exp_1), \ldots, val(exp_n)\rangle$
$\quad class =$ **case** $kind$ **of** $Nonvirtual : currClass$
$\quad\quad\quad

The auxiliary functions *invoke* and *result* manipulate the stack in a way that will be familiar to programming language implementors:

(8) $invoke(\langle val_1, \ldots, val_n \rangle, \langle var_1, \ldots, var_n \rangle, body,$
$\quad (taskStack, valStack, locStack)) \equiv$
$\quad (\langle body \rangle\ taskStack, \langle \emptyset \rangle\ valStack,$
$\quad \langle \{(var_1, val_1), \ldots, (var_n, val_n)\} \rangle\ locStack)$

(9) $result(res, (\langle _, inv \rangle\ tasks, \langle _, val \rangle\ vals, \langle _ \rangle\ locs)) \equiv$
$\quad (\langle nxt(inv) \rangle\ tasks, \langle val \oplus \{(inv, res)\} \rangle\ vals, locs)$

3.3 Exceptions

Finally, let us study the description of exception handling (in Java$_\mathcal{E}$).

The throw statement abrupts with reason $Throw(exc)$, where exc is the thrown exception:

(10) **if** $task$ is (`throw` exp `;`) **then**
$\quad$ **if** $val(exp) \neq null$ **then**
$\quad\quad mode := Throw(val(exp))$
$\quad\quad task := up(task)$
$\quad$ **else**
$\quad\quad fail(NullPointerException)$

The auxiliary function $fail$ generates an error, which is in fact treated just like an exception.

For the try-catch statement, fst, nxt, and up are defined as follows:

(11) **let** $stm = (\texttt{try}\ block\ catches)$
$\quad catches = (\texttt{catch}(\ldots)\ block_0\ \ldots\ \texttt{catch}(\ldots)\ block_n)$
in $fst(stm) = fst(block)$
$\quad nxt(block) = nxt(block_i) = nxt(stm), 0 \leq i \leq n$
$\quad up(block) = catches$
$\quad up(block_i) = up(stm), 0 \leq i \leq n$

Thus any abruption in *block* will be handled (potentially) in *catches*, whereas any abruption in *catches* itself will be handled outside the try-catch statement.

Finally, here is the semantic rule for a group of catch clauses:

(12) **if** $task$ is $(\texttt{catch}(c_0 v_0)\ b_0\ \ldots\ \texttt{catch}(c_n v_n)\ b_n)$ **then**
$\quad$ **if** $mode = Throw(exc) \wedge \exists i : 0 \leq i \leq n : catches(c_i)$ **then**
$\quad\quad loc(v_k) := exc$
$\quad\quad mode := undef$
$\quad\quad task := fst(b_k)$
$\quad\quad$ **where** $k = \iota\ i : 0 \leq i \leq n : catches(c_i)$
$\quad\quad\quad\quad \wedge \forall j : 0 \leq j < i : \neg catches(c_j)$
$\quad$ **else** $task := up(task)$
$\quad$ **where** $catches(c) = compatible(classOf(exc), c)$

If at least one of the catch clauses can handle the thrown exception ($catches(c_i)$), the first such catch clause is selected ($\texttt{catch}(c_k v_k)\ b_k$)), the thrown exception is stored in $loc(v_k)$, and control is transferred to the phrase $fst(b_k)$.

3.4 Summary

Our main observations about the ASM description are as follows:

- The ASM description spans the entire language. It is, however, incomplete in that it treats quite a large number of constructs as syntactic sugar, without taking the trouble to specify precisely how these constructs should be translated to the core constructs.
- The ASM description assumes an abstract syntax that is somewhat different from Java's concrete syntax. As well as elimination of syntactic sugar, there are many syntactic inventions, each of which encodes knowledge that would have to be inferred by a compiler.
- The ASM description is non-compositional. This was a deliberate decision by the authors, who argue that a compositional semantics is inappropriate for a concurrent programming language.
- The ASM description is very concrete and low-level, even by the standards of an operational description. The description clearly exposes stack manipulations, for example, and the control flow is explicitly encoded.
- The ASM description is moderately intelligible. The main problem is the low-level nature of the description. If a programmer wishes to understand a programming language, he/she should not be forced to understand its implementation (even an abstract implementation).
- The ASM description overloads the state with data that would better be handled explicitly by the semantic rules, such as the reason for an abruption or the current class.
- The ASM description is modular only in the limited sense that it progressively specifies a sequence of nested sub-languages.

In our view, the ASM description of Java is only moderately successful. Despite the implicit claim in the title of [3], this ASM description of Java is by no means programmer-friendly.

4 Action Semantics

In this section we study our own AS description of Java [4].

The description covers all of the Java language except threads and overloading. Moreover, it is entirely compositional, and is structured as a collection of modules. The top-level module, Semantic Functions, contains sub-modules each of which contains the semantic equations for a particular syntactic class, e.g., statements, expressions, or declarations. The module Semantic Entities contains sub-modules that specify the different sorts of data and their associated operations, e.g., primitive data, values, variables, types, classes, and objects.

4.1 Control flow

Let us now study the AS description of Java's control structures, exemplified by the while and break statements.

The semantic equation for the while statement is as follows:

(1) execute ⟦ "while" "(" E:Expression ")" S:Statement ⟧ =
    unfolding
        evaluate E then
            check (the given value is true) then
            execute S then unfold
        or
            check (the given value is false) then
            complete
    trap an unlabeled-break then complete .

This is almost the stereotypical equation for a while loop in AS. The only novelty is the use of the auxiliary combinator "_ trap _ then _", which is here used to trap an unlabeled break and then immediately complete. More generally, this auxiliary combinator:

- _ trap _ then _ :: action[escaping], reason-for-escape, action → action .

takes an action that may escape, a reason for escaping, and a second action, and performs the second action only if the first action escapes for the given reason. If the first action escapes for a different reason, the combined action also escapes for that same reason. Some of the reasons for escaping are as follows:

- reason-for-escape = break | return | throw (*disjoint*) .
- unlabeled-break : break (*individual*) .

(Note that returning from a method and throwing an exception are also reasons for escaping; these will be used in Sections 4.2 and 4.3, respectively.)

The (trivial) semantic equation for the unlabeled break statement is as follows:

(2) execute ⟦ "break" ";" ⟧ = escape with the unlabeled-break .

4.2 Methods

Let us now study the AS description of instance methods.

The semantic equation for the instance method invocation expression is as follows:

(3) evaluate ⟦ E:Expression "." I:Identifier "(" A:Arguments$^?$ ")" ⟧ =
    evaluate E and then
    respectively evaluate A
    then
        enact the application of
            the instance-method I of the class of the given object#1
            to the given (object, value*)
        or
            check there is given (null-reference, value*) then
            escape with a throw of . . . .

This evaluates the expression E, yielding the target object reference, and the tuple of arguments A. If the reference is not null, we enact the target object's instance method named I. If the reference is null, we throw a `NullPointerException`.

Equation (3) does not handle overloaded method names, where the types of the actual parameters would be used to select the correct method.

In the Java AS, a method body is represented by an abstraction of the following sort:

- instance-method = abstraction [giving a value? | ...]
  [using the given (object, value*) | ...] .

The semantic equation for the instance method declaration is as follows:

- the instance methods of ⟦ M:Modifier* R:("void" | Type) I:Identifier
  "(" F:Formal-Parameters? ")" T:Throws-Clause?
  B:Block ⟧ =
  if "static" is in the set of M then
  the empty-map
  else
  the map of the method-token of I to
  the closure of the abstraction of
  furthermore
  bind this-token to the given object#1 and
  produce the field-variable-bindings
  of the given object#1
  before
  give the rest of the given data then
  respectively formally bind F
  hence
  execute B
  trap a return then give the returned-value of it .

The declaration of an instance method results in a singleton binding of the method name to an instance-method. Note that, unlike in [2], the method body is closed with the bindings at the point of declaration. An instance-method does not use the current bindings at the point of invocation. The instance-method is given a tuple of values that consists of the target object and the actual parameters. The method body first creates the correct initial environment for the block B, which contains a binding for "`this`", the bindings for the object's fields, and bindings for the formal parameters. The block B is then executed in this environment. Finally the escape of the return is trapped and the return result is given as a transient.

The sorts of the various auxiliary operations used are as follows:

- this-token : token (*individual*) .
- class _ :: object → class (*total*) .
- field-variable-bindings _ :: object → variable-bindings (*total*) .

- instance-method _ of _ :: Identifier, class → instance-method (*partial*) .

The semantic equation for the return statement is as follows:

(4) execute ⟦ "return" E:Expression ";" ⟧ =
    evaluate E then
    escape with the return of the given value .

4.3 Exceptions

Let us now study the AS description of exceptions.

The semantic equation for the throw statement is as follows:

(5) execute ⟦ "throw" E:Expression ⟧ =
    evaluate E then
    | | check (the given reference is not null) and then
    | | escape with the throw of the given value
    | or
    | | check (the given reference is null) and then
    | | escape with the throw of the null-pointer-exception .

We evaluate E and either throw the given reference if is not null, or throw a `NullPointerException` if it is null.

The semantic equation for the try-catch statement is as follows:

(6) execute ⟦ "try" B:Block C:Catch-Clause$^+$ ⟧ =
    execute B
    trap a throw then catch in C .

We execute the block B and, if it escapes with a throw, we trap it and pass the throw to the catch-clauses C to be handled. If B escapes for a different reason, i.e., a break or return, then the escape is just propagated.

The details of handling an exception is specified by the "catch in _" operation, which is defined as follows:

(7) catch in ⟨ ⟦ "catch" "(" ⟦ T:Type I:Identifier ⟧ ")" B:Block ⟧
        C:Catch-Clause* ⟩ =
    | check (the exception of the given throw
    | is an instance of the type denoted by T) and then
    | | furthermore
    | | | allocate a variable initialized to
    | | | the exception of the given throw then
    | | | bind I to the given variable
    | | hence execute B
    or
    | check not (the exception of the given throw is
    | an instance of the type denoted by T) and then
    | catch in C .

(8) catch in ⟨ ⟩ =
escape with the given throw .

A particular catch-clause is only selected if the thrown value is an instance of the class handled by the clause (using the "_ is an instance of _" auxiliary operation). When a match is found, the thrown value is used to initialise a new variable and the binding for the catch-clause is created. The matched block B is then executed. If B itself throws an exception, it will also escape with a throw, which will be handled at an outer level. If no match for the throw is found, equation (8) propagates the throw.

4.4 Summary

In our view, the AS description is largely successful. Our main observations about it are as follows:

- The AS description is highly modular. A change in the modeling of objects, for example, would be largely localised to the module Semantic Entities/Objects; the semantic equations would hardly be affected at all.
- The AS description is highly intelligible. Anyone familiar with the terminology used in the Java Language Specification [5], should find the AS description easy to understand.
- The AS description is capable of being extended to describe threads, with only modest difficulty. In the current version of action notation (AN-1) every agent has its own local storage, so Java's shared storage would have to be handled by a shared-storage agent, and all accesses to storage ("the _ stored in _" and "store _ in _") would have to be replaced by actions to exchange messages with the shared-storage agent. In the proposed revised version of action notation (AN-2) agents share storage, so this difficulty would disappear.[1]

5 Comparison and Conclusions

In this paper we have attempted a systematic, but inevitably impressionistic, study of three formal descriptions [2–4], each of which covers a large subset of Java. When we compare the three descriptions, in terms of correctness, intelligibility, modularity, extensibility, or any other reasonable criteria, the conclusions are quite clear: the DS description is a disastrous failure, the ASM description is a partial success, and the AS description is mainly successful.

- *Correctness:* The DS description is riddled with errors, major and minor. The ASM and AS descriptions seem to be reliable, with at most a few minor errors.

[1] Of course, the whole AS description would first have to be rewritten in AN-2!

– *Intelligibility:* The DS description is practically unintelligible, even after a systematic replacement of Greek symbols with meaningful names. The ASM description is moderately intelligible, but its low-level operational nature forces the reader to understand details of the state space that are irrelevant to an understanding of the language itself. The AS description is highly intelligible, and should be accessible to ordinary Java programmers.
– *Modularity:* The DS description makes no attempt at modularity; a change in any of the semantic domains, or in the style of description (currently continuation passing), would force global changes to the semantic equations. The ASM description is modular only in the sense that it incrementally describes a sequence of nested sub-languages; a change in the state space would force global changes to the semantic rules. The AS description is highly modular. (Extending it to describe threads will be a good test of this claim.)
– *Extensibility:* The DS description is incapable of being extended to cover the whole language, since describing threads would force a change of semantic style, and consequently a complete rewrite of the existing semantic equations. The ASM description already covers the whole language, except for syntactic sugar, which probably could be added very easily. The AS description is capable of being extended to cover the whole language, with some localised changes to handle access by threads to shared storage.

Of course, it would be unreasonable to use this comparative study to draw firm conclusions about the semantic formalisms themselves. To a lesser or greater extent, there is room for improvement in all of the reviewed descriptions. It seems reasonable to conclude, however, that DS is intrinsically unsuitable for a formal description of Java: *technically*, because of its difficulty in describing concurrency, and *practically*, because of its poor intelligibility and its inability to scale up to languages of Java's complexity. ASM is harder to assess, given our limited experience with this formalism, but it too has problems with scaling up, and it seems that ASM descriptions are rather too low-level. We remain convinced that AS is the best available formalism, partly because it has just about the right level of abstractness, but also because its pragmatics encourage the writer to make the language description modular and readable. Nevertheless, AS still faces the challenge of attracting a critical mass of supporters.

References

1. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
2. J. Alves-Foss and F. S. Lam. *Dynamic Denotational Semantics of Java*, pages 201–240. In [1], 1999.
3. E. Börger and W. Schulte. *A Programmer Friendly Modular Definition of the Semantics of Java*, pages 353–404. In [1], 1999.
4. D. F. Brown and D. A. Watt. JAS: A Java action semantics. In *Proceedings of the 2nd International Workshop on Action Semantics (AS 1999)*, volume NS–99–3

of *BRICS Notes Series*, pages 43–55. Dept. of Computer Science, Univ. of Aarhus, 1999. ISSN 0909-3206.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

An Introduction to AN-2: The Proposed New Version of Action Notation

Søren B. Lassen[1]*, Peter D. Mosses[2], and David A. Watt[3]

[1] Digital Fountain, San Francisco, USA
[2] BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark
[3] Dept. of Computing Science, Univ. of Glasgow, Scotland

Abstract. Action Notation provides a suggestive *formal notation* for expressing *fundamental concepts of computation*. It is used in Action Semantics for specifying the denotations of constructs of programming languages.
This document compares the (proposed) revised version of Action Notation, AN-2, with the original Action Notation, and motivates the changes. It also gives an illustration of the use of AN-2 in an elementary action-semantic description. A complete informal description of the intended syntax and semantics of AN-2, and a formal definition of AN-2, are to be made available in separate documents.

1 Background

Action Notation (AN) is a general notation for expressing the actions that are used as semantic entities in Action Semantics. The original version of AN from 1992 (here referred to as AN-1) was described in the Action Semantics book [3], and formally defined by giving an SOS for its kernel, together with definitions of bisimulation and testing equivalences, and some laws that allowed the full AN-1 to be reduced to its kernel. AN-1 is however quite a rich notation, and its original formal definition is not easy to follow. The unconventional framework of "unified algebras" was used as a meta-notation for defining AN-1, and features of unified algebras were exploited in AN-1 itself; this may have discouraged the wider use of AN-1, and thus of Action Semantics as well.

In 1999, a modular SOS for AN-1 was provided [4, 5], confirming some suspected sources of excessive complexity in AN-1, and a revision of the design was initiated. The main aims of the revision are to *simplify the syntax and semantics* of AN, to facilitate the development of a *significantly stronger equational theory* for AN, and to *eliminate various idiosyncrasies*.

A proposal for the revised design of AN, to be known as AN-2, has been developed by the authors; a preliminary version of it (0.6) was presented at the AS 2000 workshop. The proposal included an informal summary of AN-2, some illustrative examples of the use of AN-2 in AS descriptions, and a formal

* Main contributions made while working at the University of Cambridge Computer Laboratory, supported by grant number GR/L38356 from the UK EPSRC.

definition of the syntax and semantics of AN-2. A revised version of the proposal is to be circulated on the AS mailing list for discussion between all interested parties, with the aim of reaching a consensus regarding whether or not to adopt AN-2 for future use in Action Semantics.

After recalling the design of AN-1 in Sect. 2, the current version of the tentative proposal for AN-2 is outlined and motivated in Sect. 3. An illustration of the use of AN-2 in an elementary action-semantic description is given in App. A. Some preliminary conclusions are suggested in Sect. 4.

2 The Original Action Notation, AN-1

AN-1 provides *action primitives* and *action combinators*. Action primitives may involve *yielders*, which evaluate to data depending on the information current when the primitive is performed.

AN-1 has eight parts, providing constructs for expressing the following kinds of actions:

Basic Actions: processing independently of information (action performances may terminate normally, exceptionally, fail, or diverge—perhaps non-deterministically);
Functional Actions: processing transient information (actions are given and give tuples of data);
Declarative Actions: processing scoped information (actions receive and produce bindings of tokens to data);
Imperative Actions: processing stable information (actions reserve and unreserve cells, and inspect or change the data stored in cells);
Reflective Actions: processing actions as data (abstracting actions and enacting abstractions);
Communicative Actions: processing permanent information (distributed actions performed by separate agents send messages, receive messages in buffers, and offer contracts to agents);
Directive Actions: processing scoped and stable information (actions establish indirect bindings, and redirect them to establish circular bindings); and
Hybrid Actions: involving more than one kind of information processing.

(Those parts of AN-1 concerned with just one kind of information processing were called *facets*.)

Orthogonally to the above parts, AN-1 has two *levels*:

Kernel AN-1: defined using SOS [3, App. C], and redefined using Modular SOS [4, 5];
Full AN-1: defined by laws that can be used to reduce Full AN-1 terms to Kernel AN-1 terms [3, App. B].

AN-1 is *parameterized* by a Data Notation [3, App. E] specified using Unified Algebras [3, App. F].

Grammars for the different parts of each level of AN-1 are given below in an appendix. Each construct has a reasonably clear operational interpretation [3, App. D].

3 A Tentative Proposal for AN-2

The main overall difference between the proposed revised AN-2 and the original AN-1 is that Kernel AN-2 is significantly smaller and simpler than Kernel AN-1. In particular, yielders are not included in Kernel AN-2, and are defined only in Full AN-2—essentially as abbreviations for compound kernel actions. Moreover, primitive actions do not take sorts of data as arguments, so there is no dependence on the special features of Unified Algebras in AN-2.

The treatment of bindings in AN-2 is such that although received bindings flow through sub-actions much as in AN-1, the produced bindings of AN-1 are regarded as computed values in AN-2, and simply *given as transients*. This results in a welcome simplification of the part of AN concerned with bindings: only a single combinator is needed in Kernel AN-2, and other binding combinators are defined as abbreviations in Full AN-2, using the data operations on binding maps. In AN-1, actions that give transients as well as producing bindings were very rarely used in practice, so the hybrid functional and binding combinators have been dropped altogether in AN-2..

Another simplification in AN-2 is to let actions themselves be treated as data, instead of requiring an explicit embedding as abstractions; this also eliminates the need for duplicating all the action combinators as data operations on abstractions. This change was inspired by examples of Monadic Denotational Semantics [1, 2], where the distinction between computations and computed values corresponds exactly to that between actions and data in Action Semantics.

Various other concepts have been eliminated altogether in AN-2. These include commitment, choice from arbitrary data sorts, indirect bindings, snapshots of the current storage, local storage, reuse of cells and agents, message buffers, contracting agents, message senders, message serial numbers, and offers of contracts.

The elimination of some of these concepts from AN was a high priority. For instance, commitment was often inconvenient in connection with choice between alternation actions in AN-1, and a considerable complication in its equational theory; the treatment of alternative actions in AN-2 is quite different (see the more detailed explanation below). The possibility of nondeterministic choice from sorts of cells, agents, and actions in AN-1 invalidated desirable laws that relied on locality of information. Indirect bindings were used in AN-1 merely to establish recursive closures, which is done more directly in AN-2; in any case, almost the same effect can be achieved by storing closures in ordinary cells. Snapshots of the current storage in AN-1 were potentially expensive to implement, and of little use.

In contrast, support for some of the other eliminated concepts could easily be reinstated in the Full AN-2. For instance, local use of storage could be ensured by pairing cells with the agents creating them, and checking that the agent paired with a cell matches the current agent when inspecting or updating it. Reuse of cells and agents could be allowed by storing the available ones in lists, just as received messages could be stored to give the effect of a buffer.

Contractors, senders, and serial numbers may be inserted explicitly in messages, when required.

A few concepts have been found to be lacking in AN-1, and are now supported by the proposed AN-2. These include global access to storage (appropriate when the "distributed" actions are actually threads of the same program), an "otherwise" combinator for recovering from failure, and a combinator for exceptional data sequencing. Each agent now has access to the current time, permitting the intended semantics of delay and time-out constructs to be expressed in AN-2.

To give an impression of the degree of simplification obtained in the proposed AN-2, take the size of the abstract syntax grammar for Kernel AN-2 as 1 unit. Then the size of its extension to Full AN-2 is also about 1 unit. In contrast, the size of Kernel AN-1 is almost 3 units, whereas that of its extension to Full AN-1 is less than 1 unit. Overall, Full AN-2 is little more than half the size of Full AN-1.

A thorough check will have to be made that the simplifications obtained in AN-2 have not undermined its expressiveness, nor the conciseness of action-semantic descriptions; initial experiments with taking existing descriptions written using AN-1 and reformulating them in AN-2 are, however, quite encouraging, and the degree of backwards compatibility between AN-1 and AN-2 is surprisingly high.

The rest of this section outlines the (tentative) proposed syntax for AN-2, and comments on some of the differences from AN-1. The corresponding syntax for AN-1 is given in an appendix. Note that the opportunity has been taken to replace some of the symbols used in AN-1 by new symbols that should read more naturally—for instance, "regive" has been renamed to "copy", and "escape" to "raise".

In contrast to the eight parts of AN-1, AN-2 has only five main parts, concerned with:

- Flow of Data and Control
- Scopes of Bindings
- Actions as Data
- Effects on Storage
- Interactive Processes

The separation of the flow of data and control into the functional and basic facets in AN-1 turned out not to be useful, and it has been abandoned. The other parts of AN-2 correspond closely to single parts of AN-1; the parts of AN-1 concerned with directive or hybrid binding actions have simply been eliminated.

AN-2, like AN-1, has two levels:

- *Kernel* Action Notation, defined using Modular SOS
- *Full* Action Notation, defined by reduction to Kernel Action Notation

AN-2 is also parameterized by a Data Notation that provides a collection of data sorts, operations, and predicates.

3.1 Kernel AN-2

Kernel AN-2 is intended to be as small as possible: all constructs which can be defined straightforwardly in terms of Kernel AN-2 are left to Full AN-2.

Data

$$
\begin{array}{ll}
\mathit{Data} & ::= \mathit{Datum}^* \\
\mathit{DataOp} & ::= \#i \\
\mathit{DataPred} & ::= _ = _
\end{array}
$$

DataOp consists of elements that represent operations on *Data*, including $\#i$, representing the selection of the ith component of a tuple (for each $i > 0$); similarly, *DataPred* consists of elements that represent predicates on *Data*, including $_ = _$, representing the equality of two elements of *Datum*. By using such representations of operations and predicates, the need for a higher-order meta-notation (letting data operations and predicates themselves be arguments of primitive actions) is avoided. The distinction between operations and predicates caters for frameworks (such as CASL) where predicates are regarded as primitive, and not identified with (e.g. boolean-valued) operations.

Flow of Data and Control

$$
\begin{array}{rl}
\mathit{Action} ::= & \textbf{\textit{provide}}\ \mathit{Data} \mid \textbf{\textit{copy}} \mid \mathit{Action}\ \textbf{\textit{then}}\ \mathit{Action} \mid \\
& \mathit{Action}\ \textbf{\textit{and then}}\ \mathit{Action} \mid \\
& \mathit{Action}\ \textbf{\textit{and}}\ \mathit{Action} \mid \textbf{\textit{indivisibly}}\ \mathit{Action} \mid \\
& \textbf{\textit{raise}} \mid \mathit{Action}\ \textbf{\textit{exceptionally}}\ \mathit{Action} \mid \\
& \mathit{Action}\ \textbf{\textit{and exceptionally}}\ \mathit{Action} \mid \\
& \textbf{\textit{give}}\ \mathit{DataOp} \mid \textbf{\textit{check}}\ \mathit{DataPred} \mid \\
& \textbf{\textit{fail}} \mid \mathit{Action}\ \textbf{\textit{otherwise}}\ \mathit{Action} \mid \\
& \textbf{\textit{select}}\ (\mathit{Action}\ \textbf{\textit{or}}\ \ldots\ \textbf{\textit{or}}\ \mathit{Action}) \mid \\
& \textbf{\textit{choose natural}}
\end{array}
$$

This part of Kernel AN-2 combines the Basic and Functional facets of AN-1. Notice that there are no yielders: the effect of yielder evaluation in AN-1 is achieved in Kernel AN-2 by combinations of *provide* d and *give* o, using *and* and *then*. (However, yielders are introduced, and the kernel action *give* o is extended to yielder arguments, in Full AN-2.)

For expressing normal and exceptional dataflow, AN-2 provides the same combinators as AN-1 (renaming *regive* to *copy*, *escape* to *raise*, and *trap* to *exceptionally*). It also provides the exceptional counterpart of *and then*, written A_1 *and exceptionally* A_2, where A_2 is performed—with the same data as A_1—only if A_1 terminates exceptionally, and the data given by A_1 and A_2 are concatenated only if A_2 also terminates exceptionally. For example, when

A terminates exceptionally with data *d*, *A and exceptionally raise* terminates exceptionally with *d* appended to the data given to *A*.

The treatment of alternative actions and failure in AN-2 is quite different from that in AN-1. There is now no explicit "commitment" to a selected action: storing or communicating actions do not discard alternative actions, and if a failure occurs in a selected action, an alternative action simply continues with the current information, which may well differ from that at the beginning of the first-performed alternative. However, selecting to perform an "infallible" action (i.e., one that cannot fail) corresponds to an implicit commitment, since any alternatives to it can never be performed.

AN-2 includes a new combinator, A_1 *otherwise* A_2, where A_2 is an alternative action to A_1, but not vice versa: A_2 gets performed only if A_1 fails. Symmetric choice between alternatives A_1 *or* A_2 is still available; however, all the alternatives of a particular selection now have to be enclosed in *select* (...). Thus *select*(A_1 *or* ... *or* A_n) performs any of the A_i first; on failure of the selected A_i, it performs the choice between the remaining A_j with the same data. A symmetric binary deterministic selection (where one action fails iff the other one does not) can in practice often be replaced by a simpler action using *otherwise*.

The only primitive action in AN-2 that can fail is the *fail* action itself: all other primitive actions that cannot terminate normally (e.g., due to being given inappropriate data) are supposed to terminate *exceptionally*, with no data, rather than failing. Failure is thus reserved to control choice between alternative actions, and it is expected that actions representing the semantics of most program constructs will be infallible. Note that *check p* terminates exceptionally unless *p* holds, rather than failing, so it cannot be used directly as the guard of an alternative; however, Full AN-2 defines the notation *when p* to abbreviate a corresponding action that fails unless *p* holds, so *when p then* A_1 *otherwise* A_2 expresses the deterministic selection of A_1 or A_2, depending on whether *p* holds or not.

The new treatment of alternative actions appears to be significantly simpler than the original one, both regarding the theory and the implementation of Action Notation.

In AN-1, *choose* could be used to select individual values from arbitrary sorts, including cells, agents, and actions. Thus there was no guarantee that a freshly-allocated cell or contracted agent was previously unknown, nor that knowledge of its identity could be kept local. In AN-2, *choose* is restricted so that it cannot be used for cells, agents, or actions. For simplicity (and to avoid the need for data sorts as arguments of actions) the only sort from which one may select values arbitrarily is *Natural*, the sort of natural numbers (this allows indirect selection also from sorts that have enumeration operations, e.g., sorts of characters).

Finally, note that *unfolding A* and *unfold* are no longer kernel actions in AN-2, but are regarded as abbreviations, and defined in Full AN-2. Essentially, *unfolding A* corresponds to the declaration and immediate call of a recursive

parameterless procedure with body A, and *unfold* to its recursive call, which can be expressed straightforwardly using Kernel AN-2 actions for binding and enaction.

Scopes of Bindings

Action ::= ***give current bindings*** | *Action* ***hence*** *Action*
Datum ::= *Token* | *Bindable* | *Bindings*
DataOp ::= ***binding*** | ***overriding*** | ***disjoint union***

The main difference between this part of Kernel AN-2 and the Declarative facet of Kernel AN-1 is that here, the bindings resulting from an action are simply given as transients, rather than "produced" separately from the transients. However, bindings are still "received" separately from transients, as in AN-1.

The combinator A_1 *hence* A_2 is used just as before, only now A_1 is supposed to *give* the bindings map that A_2 is to receive, and the only data given by the whole action is that given by A_2.

Various data operations are available for expressing bindings, and for combining the bindings produced by sub-actions. For instance, $binding(tk, bv)$ forms the map taking the token tk to the bindable value bv; and $overriding(b_1, b_2)$ returns the bindings map where bindings in b_2 take precedence over those in b_1 (i.e., the opposite of the operation *overlay* provided by AN-1). Most of the binding primitives and combinators of AN-1 are defined as abbreviations in Full AN-2. Note that since bindings are no longer produced separately from transients, there is no need for any hybrid functional-binding combinators at all in AN-2.

The built-in distinction of a special value *unknown*, and the AN-1 primitive action for "unbinding", are omitted in AN-2, since they were rarely used in practice in AN-1.

Actions as Data

Action ::= ***enact***
Datum ::= *Action*

This part of AN-2 corresponds to the "Reflective" part of AN-1. *Action*, the sort of actions, is a subsort of *Data* in AN-2. Thus all action combinators (and the data operation *provide_*) are automatically represented by elements of the sort *DataOp*, and can be used to compute actions—for instance, given actions A_1 and A_2 as data, performing the action *give _then_* computes A_1 *then* A_2. In AN-1, actions occurring in data had to be embedded as "abstractions", and abstraction combinators corresponding to action combinators were needed.

The action *enact* has essentially the same interpretation in AN-2 as in AN-1, except that the action to be enacted is given to it as data, rather than as an explicit argument.

The abstraction-forming data operations *provision d* and *production d* of AN-1 were mainly used to define applications and closures of abstractions. The Kernel AN-2 action *provide d* corresponds to *provision d* in AN-1; the treatment of produced bindings as data in AN-2 has allowed *produce d* to be eliminated altogether.

Effects on Storage

Action ::= ***create*** | ***destroy*** | ***update*** | ***inspect***
Datum ::= *Cell* | *Storable*

The Kernel AN-2 notation for effects on storage differs substantially from the corresponding imperative facet of AN-1. The main change is that in AN-2, cells of storage are no longer reusable: in any performance, each cell can only be given at most once by the *create* action—even if the use of the cell is subsequently stopped explicitly by the action *destroy*. In AN-1, reserved cells that became "unreserved" were regarded as no longer in the storage, and could be reserved again.

Another difference is that in AN-2, cells are always initialized when created (with a given storable value), and there is no built-in support for a distinct "uninitialized" value. Moreover, the AN-1 action "unstore" was seldom used, and has been dropped in Kernel AN-2.

Since there are no yielders in Kernel AN-2, *update* has to be given a cell and a storable value as data, and inspection of the value stored in a cell has to be an action. There is no AN-2 action corresponding to the AN-1 yielder *current storage*, which has not been of much use in practice.

A further difference between the treatment of storage in AN-2 compared to that in AN-1 is that storage is in principle *global*, and cells created by one agent may be updated, inspected, or even destroyed by other agents.

Interactive Processes

Action ::= ***activate*** | ***deactivate*** | ***give current agent*** |
send | ***receive*** | ***give current time***
Datum ::= *Agent* | *Message* | *MessageTag*

This part of Kernel AN-2 corresponds to the Communicative facet of AN-1. When given an action as data, *activate* adds a new agent performing that action to the system, and gives the identity of the agent; this has a similar effect to offering a contract in AN-1, and waiting for the action of the contract to send back a message revealing its performing agent. The action *deactivate*, which (eventually) stops the performance of a given agent, is included mainly because of the difficulty of adding it later as an abbreviation (but it is presently unclear whether it is sufficiently useful to warrant its inclusion in AN-2 at all).

Sending a message in AN-2 is the same as in AN-1, except that the sender of the message is not implicitly included (so anonymous messages are now possible). The action to receive a message in AN-2, however, does not wait patiently for an appropriate message to arrive: it simply terminates exceptionally if such a message is not already available. A further difference is that the AN-1 buffer of messages that have arrived at an agent, but which have not yet been dealt with, is left implicit in AN-2. Moreover, discrimination between different kinds of messages in AN-2 is purely on the basis of *message tags* provided by the senders, instead of by specifying an entire sort of messages.

A new feature of AN-2 is that each agent has access to the (local) current time. Various data operations will be provided to select the year, month, hour, minute, second, millisecond, etc., from the time, as well as for addition and subtraction of times. Even though nothing is assumed about the (absolute or relative) speeds of the agents that perform actions, it seems both desirable and realistic to allow delaying, i.e. checking the current time until it exceeds some absolute value before proceeding. (It would be possible to represent clocks by agents, but it may then involve an excessive amount of communication to obtain the time—and when there is more than one clock agent, an elaborate protocol would be needed to keep the different clocks "on time".)

As mentioned above, storage is no longer local to particular agents. Since buffers, explicit serial numbers, and information about contracting agents have been eliminated in AN-2, the representation of an agent performing an action is much simpler than it was in AN-1.

3.2 Extension to Full AN-2

To extend Kernel AN-2 to Full AN-2 we introduce values representing subsorts of *Data*, we introduce yielders, and we define many of the remaining AN-1 action primitives and combinators as straightforward abbreviations.

Data

DataOp ::= ***the*** *DataSort* | ***a*** *DataSort* | ***an*** *DataSort* | ***it***

DataSort includes elements representing all subsorts of *Data*, written as the lowercase spelling of the sort symbols. For any *s* in *DataSort*, the data operation *the s* is defined as the projection from *Data* to the subsort represented by *s* (the result being the argument when it is in the subsort, otherwise undefined). The data operations *a s* and *an s* are the same as *the s*, but intended for use when first referring to some given data (as in English). The data operation *it* abbreviates *the datum*, being defined only on 1-tuples.

Flow of Data and Control

$$
\begin{array}{lll}
\mathit{Yielder} & ::= & \mathit{Data} \mid \mathit{DataOp} \mid \mathit{DataOp}\ \ \mathit{Yielder} \mid (\mathit{Yielder}, \ldots, \mathit{Yielder}) \\
\mathit{Enquirer} & ::= & \mathit{DataPred} \mid \mathit{DataPred}\ \ \mathit{Yielder} \\
\mathit{Action} & ::= & \textbf{\textit{give}}\ \ \mathit{Yielder} \mid \mathit{Action}\ \ \mathit{Yielder} \mid \\
 & & \textbf{\textit{given}}\ \ \mathit{Yielder} \mid \textbf{\textit{when}}\ \ \mathit{Enquirer} \mid \\
 & & \textbf{\textit{skip}} \mid \textbf{\textit{err}} \mid \textbf{\textit{tentatively}}\ \ \mathit{Action} \mid \textbf{\textit{infallibly}}\ \ \mathit{Action}
\end{array}
$$

Data operations occurring in yielders are applied either to the current given data, or to explicit yielder arguments. For instance, the yielder *the s* refers to the given data, and projects it onto the subsort s, thus corresponding to *given s* in AN-1. The compound yielder *the s Y* in AN-2 corresponds to the AN-1 yielder *the s yielded by Y*, so in particular, *the s#i* projects the ith component of the data onto the subsort s, and corresponds to *given s#i* in AN-1. All components of a tuple yielder $(Y_1, \ldots, Y_n)$ are required to yield elements of *Datum*.

As may be expected, the action *give Y* gives the data yielded by Y. The action *A Y* merely abbreviates *give Y then A*. Typically, A here will be a simple action (such as *update*, *inspect*, or *raise*) that expects to be given certain data, and Y will be a yielder that computes the expected data. An example is *update* (*the cell#2*, *the storable#1*). If the expected data is already available, the insertion of Y may still be useful for emphasis (or for fluency of reading), as in *inspect the cell*. Notice that the compound action *raise Y* expresses the same as the AN-1 action *escape with Y*.

In AN-2, *given Y* is an action, and intended for use as a guard: when the given data is identical to the data yielded by Y, it simply copies the given data, but otherwise it *fails*. Thus *given d* tests that the given data is the same as d, and *given an s* tests that the given data is in the subsort represented by s.

An enquirer is a generalization of data predicates to allow their composition with yielders. The action *when Q* checks whether the enquirer Q holds or not, copying the arguments of the predicate if it does, but failing otherwise.

The remaining actions are all rather trivial: *skip* abbreviates *provide* (); *err* abbreviates *provide* () *then raise*; *tentatively A* fails whenever A terminates exceptionally with no data, and vice versa for *infallibly A*.

Scopes of Bindings

$$
\begin{array}{lll}
\mathit{Yielder} & ::= & \textbf{\textit{current bindings}} \mid \textbf{\textit{bound to}}\ \ \mathit{Yielder} \mid \\
 & & \textbf{\textit{closure}}\ \ \mathit{Yielder} \\
\mathit{Action} & ::= & \textbf{\textit{bind}} \mid \textbf{\textit{furthermore}}\ \ \mathit{Action} \mid \\
 & & \mathit{Action}\ \ \textbf{\textit{moreover}}\ \ \mathit{Action} \mid \mathit{Action}\ \ \textbf{\textit{before}}\ \ \mathit{Action} \mid \\
 & & \textbf{\textit{recursively}}\ \ \mathit{Action} \mid \textbf{\textit{unfolding}}\ \ \mathit{Action} \mid \textbf{\textit{unfold}}
\end{array}
$$

The AN-2 yielder *bound to Y* is equivalent to the AN-1 yielder *the bindable bound to Y*. The yielder *the s bound to Y* has the same meaning in both AN-1

and AN-2, but is parsed in AN-2 as *the s* (*bound to Y*), where *the s* is the data operation projecting onto the subsort corresponding to *s*.

The action *bind* may be written with arguments as *bind* (*tk*, *Y*), corresponding closely to the AN-1 action *bind tk to Y*. The binding combinators *furthermore*, *moreover*, and *before* are defined to give essentially the same effect as in AN-1.

Recursion was provided only for single bindings in AN-1, and mutual recursion had to be expressed using explicit indirect bindings. Despite the close relationship of indirect bindings to "forward" declarations in programming languages, it is preferable to provide a general combinator for mutual recursion, and this has now been achieved in the proposed AN-2: *recursively A* is as *A*, except that *recursively A* is also inserted (appropriately) in any closures formed when performing *A*. In particular, *recursively bind*(*tk*, *closure A′*) is equivalent to

bind tk to closure (*furthermore* (*recursively bind*(*tk*, *closure A′*))
hence A′)

and similarly for mutually-recursive bindings to closures. Note however that if *A* has effects on storage or interactions with other processes, these get repeated every time any closure in *recursively A* is enacted; in general, to avoid unexpected consequences, *A* should be formed only from dataflow and binding actions.

The Kernel AN-1 notation for unfolding is provided merely as an abbreviation in Full AN-2, but the usage and intended interpretation remain the same.

Actions as Data

In AN-2, the AN-1 yielder *application* Y_1 *to* Y_2 can be written as the yielder (*provide* Y_2) *then* Y_1, so there seems to be no need for the former notation in AN-2. Notice that here, *provide* Y_2 is not an action but an action-yielder, composing the data operation *provide_* (which maps data to actions) with the yielder Y_2.

Effects on Storage

Yielder ::= ***stored in*** *Yielder*

The AN-2 yielder *stored in Y* is equivalent to the AN-1 yielder *the storable stored in Y*. The yielder *the s stored in Y* has the same meaning in both AN-1 and AN-2, but is parsed in AN-2 as *the s* (*stored in Y*), where *the s* is the data operation projecting onto the subsort corresponding to *s*.

The action *update* may be written with arguments as *update* (Y_1, Y_2), corresponding exactly to the AN-1 action *store* Y_2 *in* Y_1, so there seems to be no need for the latter notation in AN-2.

Interactive Processes

Yielder ::= ***current agent*** | ***current time***
Action ::= ***patiently*** *Action*

Note that the identity of the contracting agent is not provided in AN-2. The combinator *patiently A* abbreviates *unfolding* (*A otherwise unfold*); busy waiting for a message to arrive can then be expressed as *patiently tentatively receive*.

4 Conclusion

The proposed revised version of Action Notation, AN-2, is substantially smaller than the original version, yet manages to retain much of its expressiveness. AN-2 does not rely on any special features of the formalism used to specify data; in particular, familiarity with Unified Algebras is no longer a prerequisite for understanding Action Notation. The smaller size of Kernel AN-2 should be especially advantageous for the development of a strong equational theory for Action Notation, as as for those providing tool support for Action Semantics.

It should be straightforward to convert existing action-semantic descriptions to use AN-2. More work would be needed to convert existing tools for Action Semantics, but the change to a new version should also encourage a greater degree of uniformity between tools regarding the details of the syntax and semantics of the action notation that they accept.

Those working on or with Action Semantics are asked to take a close critical look at the details of the proposed revised version [6], and let the authors know about any problems. It is hoped to reach a consensus by the end of year 2000 regarding the desirability of adopting AN-2 for future use in Action Semantics.

References

1. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *LNCS*, pages 219–234. Springer-Verlag, 1996.
2. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
3. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
4. P. D. Mosses. A modular SOS for Action Notation. Research Series RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [5].
5. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In P. D. Mosses and D. A. Watt, editors, *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at `http://www.brics.dk/RS/99/56/`.
6. P. D. Mosses. AN-2: Revised action notation—informal summary. Available at `http://www.brics.dk/~pdm/`, Sept. 2000.

A Illustrative Action-Semantic Description

The action-semantic description given below illustrates the use of most of the full AN-2, apart from that concerned with interactive processes. (The meta-notation used is experimental, and leaves implicit the modular structure of the description.)

A.1 Abstract Syntax

- *Expression* ::= *Literal* | *UnaryOperator Expression* |
  Expression BinaryOperator Expression | *Identifier*
- *Literal* ::= *Natural* | *Boolean*
- *UnaryOperator* ::= `-` | `!`
- *BinaryOperator* ::= `+` | `-` | `*` | `==` | `<` | `>`
- *Identifier* ::= *String*
- *Statement* ::= *Block* | *Identifier* `=` *Expression* `;` |
  `if(`*Expression*`)`*Statement* `else` *Statement* |
  `while(`*Expression*`)`*Statement* | `break;` |
  Statement `or` *Statement* | *Identifier*`();`
- *Block* ::= { } | { *Statement Statement* } | { *Declaration Statement* }
- *Declaration* ::= *Type Identifier* `=` *Expression* `;` | `void` *Identifier*`()` *Block*
- *Type* ::= `int` | `boolean`

A.2 Semantic Functions

- *evaluate*⟦_⟧ : *Expression* → *Action*
  %{ normally giving a value **}%**
- *operate*$_1$⟦_⟧ : *UnaryOperator* → *Action*
  %{ given a value, normally giving a value **}%**
- *operate*$_2$⟦_⟧ : *BinaryOperator* → *Action*
  %{ given (a value, a value), normally giving a value **}%**
- *execute*⟦_⟧ : *Statement* → *Action*
  %{ normally giving (), exceptionally giving a break **}%**
- *declare*⟦_⟧ : *Declaration* → *Action*
  %{ normally giving a bindings **}%**

A.3 Semantic Entities

AN-2 **version** 0.7.3
DN-2 **%{** forthcoming **}%**

- *Datum* ::= *Value* | *Break*
- *Bindable* ::= *Variable* | *Procedure*
- *Storable* ::= *Value*
- *Value* ::= *Integer* | *Boolean*
- *Variable* ::= *Cell*
- *Break* ::= *break*
- *Procedure* ::= *Action*

A.4 Semantic Equations

E, E_1, E_2 : *Expression*; L : *Literal*; UO : *UnaryOperator*; BO : *BinaryOperator*;
I : *Identifier*; S, S_1, S_2 : *Statement*; D : *Declaration*; T : *Type*;

Expressions

- $evaluate[\![L]\!] = give\ L$
- $evaluate[\![UO\ E]\!] = evaluate[\![E]\!]\ then\ operate_1[\![UO]\!]$
- $evaluate[\![E_1\ BO\ E_2]\!] = (evaluate[\![E_1]\!]\ and\ evaluate[\![E_2]\!])$
  $then\ operate_2[\![BO]\!]$
- $evaluate[\![I]\!] = inspect\ the\ variable\ bound\ to\ I$

Unary Operators

- $operate_1[\![\texttt{-}]\!] = give\ (-the\ integer)$
- $operate_1[\![\texttt{!}]\!] = give\ (not\ the\ boolean)$

Binary Operators

- $operate_2[\![\texttt{+}]\!] = give\ (the\ integer\#1 + the\ integer\#2)$
- $operate_2[\![\texttt{-}]\!] = give\ (the\ integer\#1 - the\ integer\#2)$
- $operate_2[\![\texttt{==}]\!] =$
  $select(when\ (the\ integer\#1 = the\ integer\#2)\ or$
  $when\ (the\ boolean\#1 = the\ boolean\#2))$
  $then\ give\ true\ otherwise\ give\ false$

- $operate_2[\![\texttt{<}]\!] =$
  when (*the integer*#*1* $<$ *the integer*#*2*)
  then give true otherwise give false

- $operate_2[\![\texttt{>}]\!] =$
  when (*the integer*#*1* $>$ *the integer*#*2*)
  then give true otherwise give false

Statements

- $execute[\![I \texttt{ = } E \texttt{;}]\!] =$
  (*give the variable bound to* I *and evaluate*$[\![E]\!]$)
  then update

- $execute[\![\texttt{if(}E\texttt{)}S_1 \texttt{ else } S_2]\!] =$
  evaluate$[\![E]\!]$ *then infallibly select*(
  (*given true then execute*$[\![S_1]\!]$) *or*
  (*given false then execute*$[\![S_2]\!]$))

- $execute[\![\texttt{while(}E\texttt{)}S]\!] =$
  unfolding(
  evaluate$[\![E]\!]$ *then infallibly select*(
  (*given true then execute*$[\![S]\!]$ *and then unfold*) *or*
  (*given false then skip*)))
  exceptionally (*given break then skip otherwise raise*)

- $execute[\![\texttt{break;}]\!] =$ *raise break*

- $execute[\![I\texttt{();}]\!] =$ *enact the procedure bound to* I

Blocks

- $execute[\![\texttt{\{ \}}]\!] =$ *skip*

- $execute[\![\texttt{\{ }S_1\ S_2\texttt{ \}}]\!] = execute[\![S_1]\!]$ *and then* $execute[\![S_2]\!]$

- $execute[\![\texttt{\{ }D\ S\texttt{ \}}]\!] =$ *furthermore* $declare[\![D]\!]$ *hence* $execute[\![S]\!]$

Declarations

- $declare[\![T\ I \texttt{ = } E \texttt{;}]\!] =$
  evaluate$[\![E]\!]$ *then create then bind*$(I,$ *the variable*$)$

- $declare[\![\texttt{void } I\texttt{() } B]\!] =$
  recursively bind$(I,$ *the procedure closure execute* $B)$

B The Syntax of the Original Action Notation, AN-1

B.1 Kernel AN-1

Note that *Data* in AN-1 includes not only individual values but also proper subsorts of values (exploiting the treatment of sorts in Unified Algebras).

Basic

Action ::= ***complete*** | ***escape*** | ***fail*** | ***commit*** | ***unfold*** |
unfolding *Action* | ***indivisibly*** *Action* |
Action ***or*** *Action* | *Action* ***and*** *Action* |
Action ***and then*** *Action* | *Action* ***trap*** *Action*
Yielder ::= ***the*** *Data* ***yielded by*** *Yielder* |
dataop(*Yielder*, . . . , *Yielder*)

Functional

Action ::= ***give*** *Yielder* | ***choose*** *Yielder* | *Action* ***then*** *Action*
Yielder ::= ***them***
Data ::= *Datum**

Declarative

Action ::= ***bind*** *Yielder* ***to*** *Yielder* | ***unbind*** *Yielder* |
produce *Yielder* | *Action* ***moreover*** *Action* |
Action ***hence*** *Action* | *Action* ***before*** *Action*
Yielder ::= ***current bindings*** |
Yielder ***receiving*** *Yielder*
Data ::= *Tokens* | *Bindable* | *Bindings* |
unknown | ***known*** *Data*

Imperative

Action ::= ***store*** *Yielder* ***in*** *Yielder* | ***unstore*** *Yielder* |
reserve *Yielder* | ***unreserve*** *Yielder*
Yielder ::= ***current storage***
Data ::= *Cell* | *Storable* | *Storage* |
uninitialized | ***initialized*** *Data*

Reflective

Action ::= ***enact*** *Yielder*
Data ::= *Abstraction* | ***abstraction of*** *Action* |
actionop(*Data*, . . . , *Data*) |
provision *Data* | ***production*** *Data*

Communicative

Action ::= ***send*** *Yielder* | ***remove*** *Yielder* |
offer *Yielder* | ***patiently*** *Action*
Yielder ::= ***current buffer*** | ***performing agent*** |
contracting agent
Data ::= *Agent* | *Buffer* | *Communication* | *Message* | *Sendable* |
Contract | ***user agent*** | ***contents*** *Data* |
sender *Data* | ***receiver*** *Data* | ***serial*** *Data* |
Data ***[containing*** *Data]* | *Data* ***[from*** *Data]* |
Data ***[to*** *Data]* | *Data* ***[at*** *Data]*

Directive

Action ::= ***indirectly bind*** *Yielder* ***to*** *Yielder* |
redirect *Yielder* ***to*** *Yielder* |
undirect *Yielder* | ***indirectly produce*** *Yielder*
Yielder ::= ***current redirections***
Data ::= *Indirection* | *Redirections* | ***indirect production*** *Data*

Hybrid

Action ::= *Action* ***and then moreover*** *Action* |
Action ***then moreover*** *Action* |
Action ***thence*** *Action* | *Action* ***then before*** *Action*
Data ::= ***owner*** *Data* | *Data* ***[on*** *Data]*

B.2 Extension to Full AN-1

Basic
(None)

Functional

Action ::= ***escape with*** *Yielder* | ***regive*** | ***check*** *Yielder*
Yielder ::= ***given*** *Yielder* | ***given*** *Data* # *Pos* | ***it***

Declarative

Action ::= ***rebind*** | ***furthermore*** *Action*
Yielder ::= ***the*** *Data* ***bound to*** *Yielder*

Imperative

Yielder ::= ***the*** *Data* ***stored in*** *Yielder*

Reflective

Yielder ::= ***application*** *Yielder* ***to*** *Yielder* | ***closure*** *Yielder*

Communicative
(None)

Directive

Action ::= ***recursively bind*** *Yielder* ***to*** *Yielder*
Yielder ::= ***indirect closure*** *Yielder*

Hybrid

Action ::= ***allocate*** *Yielder* | ***receive*** *Yielder* | ***subordinate*** *Yielder*

Postfix Transformations for Action Notation

Kent D. Lee

Luther College
leekentd@luther.edu

Abstract. Actions are a relatively high-level description of computation when compared to assembly language. If compilers are to be automatically generated from action semantic descriptions of programming languages, some transformations will be necessary to simplify the resulting actions and make them suitable for code generation in a target architecture. Hermano Moura studied action transformations in his PhD thesis. This paper describes a different approach to action transformation that is desirable when generating code for RISC and stack-based architectures.

1 Introduction

Action Semantics is a formal language for the description of programming languages. Actions describe computations in terms of the manipulation of transient data, bindings of identifiers to data, and a store mapping cells to data. For a complete description of Action Semantics, see [6]. As a formal language, it is possible to generate compilers automatically from Action Semantic descriptions of programming languages. Action Semantics directed compiler generation has been studied by several groups [2][3][8].

Figure 1 illustrates the structure of the Genesis compiler generator, which is similar in structure to other Action Semantics-based compiler generators. The resulting compilers translate a source program into a program action. A program action represents the computation of a particular source language program. In the final stage of the generated compilers, the program action is given to a code generator, which translates the program action to a target program. In the case of Genesis, the target language is Java byte code for the Java Virtual Machine.

2 Action Transformation

Action transformation is the process of simplifying an action for two purposes. The first goal is to enable efficient code to be generated from the action. A second goal is to make the code generation stage as simple as possible. Hermano Moura[7] studied action transformations with the goal of eliminating as many bindings and transients as possible to satisfy the first goal above while using the C programming language as the target to satisfy the second goal [2]. For purposes of this paper, consider the Small programming language (a subset of

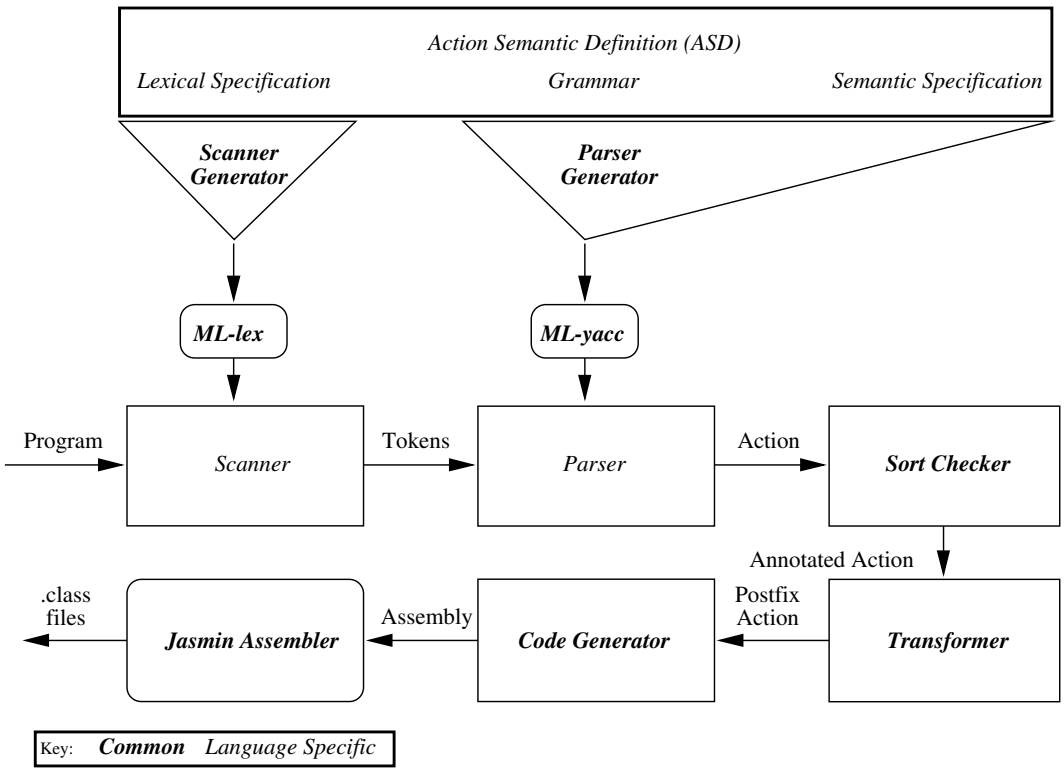


Fig. 1. structure of an Action Semantics-based Compiler Generator

ML). The Small program given below is described by the program action given in figure 2, which is derived from the Action Semantic description for the Small language given in [3].

```
let val i=5+4
in
  output(i)
end
```

Moura describes an action transformation algorithm in his thesis[7]. Using Moura's algorithm, after binding elimination, the action in figure 2 may be simplified to the action given in figure 3. After further performing transient elimination on the action, the action in figure 4 is derived. Notice that there are still transients in the residual action. The sum of 4 and 5 is not known statically because in this case the action transformer does not know how to compute the sum of integers. If the transformer were a partial evaluator it could carry out the sum operation on the static data. Partial evaluation of actions has been studied in [1]. Introducing partial evaluation in the transformer creates other problems that must be solved. Assuming that the transformer is not a partial evaluator, the problem that this paper addresses is how to transform an action to get efficient target programs while keeping the code generator simple.

give 5
and then
give 4
then
give the sum of the given data
then
bind “i” to the given integer
hence
give the integer bound to “i”
then
enact application of the native abstraction bound to “output” to the given data

Fig. 2. program action for a Small program

give [integer]cell$_{(0,0)}$
and then
give 5
and then
give 4
then
give the sum of the given data
then
store the given integer#2 in the given [integer]cell#1
hence
give [integer]cell$_{(0,0)}$
then
give the integer stored in the given [integer]cell
then
enact application of the native abstraction$_{output}$ to the given data

Fig. 3. program action after binding elimination

give the sum of (5,4)
then
store the given integer in [integer]cell$_{(0,0)}$
hence
give the integer stored in [integer]cell$_{(0,0)}$
then
enact application of the native abstraction$_{output}$ to the given data

Fig. 4. after performing transient elimination on the action in figure 3

3 Code Generation

To keep code generation simple, it is desirable to transform actions to as small a subset of action notation as possible. The transformations performed to arrive at the action in figure 4 result in inconsistency in the residual action. Consider the Small program given below.

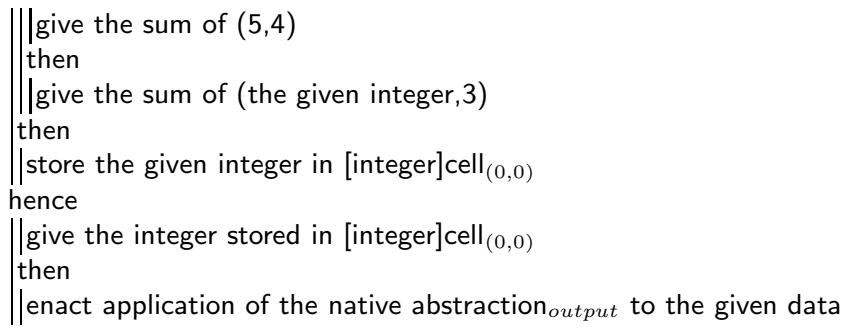


Fig. 5. an action that inconsistently gives transients

```
let val i=5+4+3
in
  output(i)
end
```

After performing the transformations on the program action for this program we arrive at the action given in figure 5. The code generator would have to be written to generate code for summing integers where both integers are known and where only one integer is known while the other is given as a transient. This results in three different cases for generating code for summing integers. However, if transients were not eliminated, the residual action for the first program would look like the one given in figure 3. While transients are left in the residual action in figure 3, it is consistent with respect to how transients are handled within the action. Moreover, the action closely resembles low-level code that would be generated for this program on both stack-based and RISC architectures. The Java Virtual Machine is one such machine. Jasmin is an assembler of Java byte code [5]. A excerpt from a Jasmin assembly language program for the action given above is

```
.method public run_Main()V
    .throws java/io/IOException
    .limit stack 4
    .limit locals 1
    aload_0
    getfield test25_Frame/localData [I
```

```
    iconst_0
    iconst_5
    iconst_4
    iadd
    iastore
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload_0
    getfield test25_Frame/localData [I
    iconst_0
    iaload
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

Notice the similarity between this program and the action presented in figure 3. In this program the address for the given cell is pushed on the stack first, followed by the 5 and 4. Then the sum of 5 and 4 is computed and the result is stored in the given cell. This is nearly identical to the performance of the action. In a stack-based architecture the stack holds transient values given by actions.

RISC architectures are similar to stack-based architectures in many ways if you use a simple register allocation scheme. A framework for one such allocation scheme is presented in [4]. An excerpt from a MIPS assembly language program for the given action is

```
move $t0, $sp
li $t1, 5
li $t2, 4
add $t1,$t1,$t2
store $t1,0($t0)
```

The code above follows a simple demand based register allocation scheme. There is support in the literature for just such an allocation scheme [9]. This demand allocation scheme emulates a stack using the general purpose registers of the RISC architecture.

4 Postfix Actions

This paper proposes the definition of a postfix form for actions. The action given in figure 3 is in postfix form. An action is in postfix form if

- all datum that are used in the action are given (i.e. Individuals never yield themselves except in the context of an action that gives them).
- transients may not be regiven.

The first criterion insures that all datum appear as transients. This is important during code generation because transients are equivalent with the contents of

the stack in a stack-based architecture and the contents of registers in RISC architectures. By requiring every datum to appear as a transient in the action, the datums are likewise required to appear on the stack or in a register during execution on the target architecture.

The second criterion preserves the first in/first out nature of a stack. When a datum is given by a postfix action it corresponds to pushing it on a stack in a stack-based architecture. The goal of the postfix form is to allow action performance using a stack to hold the transient values. If an action regives a datum that is not on the top of the stack it would violate the first in/first out stack property, which means a stack is no longer a sufficient data structure to hold the transient data. Therefore, postfix actions are just those actions where a stack is sufficient to hold the transient values during their performance. On RISC architectures, the register allocation framework presented in [4] requires that registers be allocated in a first in/first out-like fashion as well, so the postfix form for actions applies to RISC architectures as well.

5 Action Transformation for Postfix Actions

Actions that are generated by a grammatical interpretation of a source program generally give their data as transients as required by the postfix form described above. Therefore, transient elimination is not generally needed during action transformation for postfix actions. Binding elimination is still needed to reduce the overhead of creating bindings at run-time. The transformation algorithm for postfix actions proceeds by eliminating bindings in a bottom-up fashion. Actions like

bind "*id*" to y

where y yields an unknown value may be replaced with

||give [*datum_type*]cell$_i$
and then
||give y
then
|store the given *datum_type*#2 in the given [*datum_type*]cell#1

where *datum_type* is a sort containing the *datum* yielded by y. Subsequent references to the eliminated binding can be transformed to refer to the contents of the named cell, [*datum_type*]cell$_i$, instead. Notice that the bind action is replaced by an action that seems to adhere to the postfix property. Unfortunately, this transformation does not preserve the postfix property of the action. For instance, consider the action given in figure 2. After applying the binding elimination transformation above, the action in figure 6 is obtained. But, this action contains an occurrence of give the given integer which violates the second postfix condition above. To fix this a relatively complex transformation has to be applied. If an action of the form

|||give 5
|and then
|||give 4
|then
||give the sum of the given data
then
|||give [integer]cell$_{(0,0)}$
|and then
|||give the given integer
|then
||store the given integer#2 in the given [integer]cell#1
hence
|||give [integer]cell$_{(0,0)}$
|then
||give the integer stored in [integer]cell$_{(0,0)}$
then
||enact application of the native abstraction$_{output}$ to the given data

Fig. 6. program action for a Small program

|a_1
then
|||give y
|and then
|||give the given s
|then
||a_2

is encountered and y does not refer to transients or the contents of a cell, then it may be transformed to

||give y
|and then
||a_1
then
|a_2

The action given in figure 3 was derived by applying this transformation to the action given in figure 6. It is possible to apply a slightly different transformation for binding elimination. Bind actions like the one above may be replaced with the action

||give y
|and then
||give [$datum_type$]cell$_i$
then
|store the given $datum_type$#1 in the given [$datum_type$]cell#2

This transformation leads to a similarly malformed action requiring another complex transformation to keep the action in postfix form. The complex transformation replaces actions of the form

```
|a1
then
|||give the given s
|and then
|||give y
|then
||a2
```

with the action

```
||a1
|and then
||give y
|then
|a2
```

assuming that y does not refer to transient data.

```
||give 5
|then
||bind "i" to the given integer
hence
||give the integer bound to "i"
|then
||enact application of the native abstraction bound to "output" to the given data
```

Fig. 7. program action for a Small(er) program

```
||give 5
|then
||complete
hence
||give 5
|then
||enact application of the native abstraction bound to "output" to the given data
```

Fig. 8. program action after applying binding elimination to eliminate a known value

Finally, bind actions that bind identifiers to known values may be replaced by the action complete. Consider the (even smaller) Small program

```
let val i = 5
in
  output(i)
end
```

The program action for this program is given in figure 7. In this case, "i" is bound to a known value and a cell is not needed to store a known value. The bind action may be replaced by complete and references to the bound value may be replaced with the value itself. After applying these transformations the action in figure 8 is derived.

$consume$: action$\rightarrow$ action
$consume\ a_1$ and $a_2 = consume\ a_1$ and $consume\ a_2$
| $consume\ a_1$ and then $a_2 = consume\ a_1$ and then $consume\ a_2$
| $consume\ a_1$ hence $a_2 = consume\ a_1$ hence $consume\ a_2$
| $consume\ a_1$ moreover $a_2 = consume\ a_1$ moreover $consume\ a_2$
| $consume\ a_1$ before $a_2 = consume\ a_1$ before $consume\ a_2$
| $consume\ a_1$ else $a_2 = consume\ a_1$ else $consume\ a_2$
| $consume\ a_1$ or $a_2 = consume\ a_1$ or $consume\ a_2$
| $consume\ a_1$ then $a_2 = a_1$ then $consume\ a_2$
| $consume\ a_1$ thence $a_2 = a_1$ thence $consume\ a_2$
| $consume$ give $y =$ complete
| $consume$ allocate $y =$ complete
| $consume\ a = a$

Fig. 9. The *consume* Operation

|give 5
then
|enact application of the native abstraction bound to "output" to the given data

Fig. 10. program action after consuming unneeded transients and applying identities

The action in figure 8 now gives transients that are not needed. While this does not violate the postfix property of actions, it is unnecessary and could lead to extra garbage being pushed onto the stack of a stack-based architecture. To eliminate unneeded transients, an operation called *consume* may be applied to an action, a, when discovered in the context of

|a
then
|complete

or when it appears in the context of

$|a$
thence
|complete

The consume operation is defined in figure 9. After consuming unneeded transients, identities involving complete may have to be applied to further simplify the action. After consuming the unneeded transients in the action in figure 8 and applying the identity transformations the result is the action in figure 10.

6 Conclusion

Moura presented a transformation algorithm that was suitable for targeting higher level languages like C as targeted in the Actress compiler generator. When targeting lower level languages, issues like the contents of the stack or registers become more important.

This paper has presented a definition of postfix actions and motivated their use during the code generation phase of Action Semantics directed compilers targeting stack-based and RISC architectures. The two goals of efficient target programs and simple code generation have been addressed in the context of action transformations that preserve the postfix property of actions.

Work that remains in this area involves extending Genesis and other Action Semantics directed compiler generators to support a larger subset of action notation and/or to support newer versions of action notation.

References

1. A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FCPA '93)*, Copenhagen, DK, 1993.
2. D.F. Brown, H. Moura, and D.A. Watt. Actress: an action semantics directed compiler generator. In *Proceedings of the Workshop on Compiler Construction*, Paderborn, Germany, 1992.
3. K.D. Lee. *Action Semantics-based Compiler Generation.* PhD thesis, Department of Computer Science, University of Iowa, 1999.
4. K.D. Lee. Minimal register allocation. Technical Report 99-06, University of Iowa, Department of Computer Science, Iowa City, IA, 1999.
5. J. Meyer. Jasmin. WWW Home Page, Media Research Laboratory, New York University, March 1997. URL: `http://www.cat.nyu.edu/meyer/jasmin`.
6. P.D. Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26.* Cambridge University Press, 1992.
7. H. Moura. *Action Notation Transformations.* PhD thesis, Department of Computer Science, University of Glasgow, 1993.
8. P. Ørbæk. Oasis: An optimizing action-based compiler generator. In *Proceedings of the International Conference on Compiler Construction, Volume 786*, Edinburgh, Scotland, 1994. LNCS.
9. T. Pittman and J. Peters. *The Art of Compiler Design.* Prentice Hall, Englewood Cliffs, NJ 07632, 1992.

Action Semantics for Logic Programming Languages

Luis Carlos Menezes*,
Hermano Perrelli de Moura, Geber Ramalho

Centre of Informatics
Federal University of Pernambuco
CP 7851, CEP 50732-970, Recife, Brazil.
E-mail: {lcsm,moura,glr}@cin.ufpe.br.

Abstract. Logic programming languages have some particular concepts which require the definition of complex structures if one tries to give a formal semantics to them. This complexity difficulties the writing of formal descriptions and makes the resultant specifications too complex and hard to be read and understood by others. This paper proposes to extend action semantics with some new semantic entities to represent these concepts. Using these entities, the description of logic programming languages becomes easy and the produced specifications are more readable. We expect that it could also improve the use of formal methods in the design of logic programming languages and could help in the definition of new languages in this paradigm.

1 Introduction

Action semantics defines facets to describe the most common concepts found in programming languages (like values, functions, bindings, storage, sequencing, parallelism, interleaving). Using these facets the description of programming languages like Pascal, C and ML becomes easier.
The logical programming paradigm facilitates the design of complex programs because their structures, based in first order logic, frees the programmer to handle directly with lower level control structures like the existing in imperative programming languages.
When we work in formal description of logic programming languages, the facets of action semantics are not helpful because they do not support some complex concepts used by these languages (relations, backtracking, unifications, etc). The produced specifications are too complex and hard to manipulate.
To reduce this problem, we propose to improve the support of these concepts in action semantics. This paper shows some new primitive actions and combinators to represent the control flow and data structures found in languages like Prolog. Using these new actions, the specifications becomes shorter and more readable.
This text is organized in the following way:

* Supported by CNPq, Brazil.

– the first section shows the properties of logic programming languages;
– the second and third sections justify and present an extension of action notation, designed to represent the properties described in the first section;
– the fourth section exemplifies the use of the proposed extension giving a semantics for the Prolog programming language;
– the last section analyses the results obtained with the semantics entities proposed in this paper.

2 Logic Programming Languages

```
parent(Peter, John).
parent(Wally, Fred).
parent(John, Al).

Question :- parent(X,Y),parent(Y,Z).
```

Fig. 1. Small example of a logic program.

The logic paradigm uses first order logic to specify the programs behavior. The most important structure in a logic program is the first order predicate, formed by a predicate name and a set of arguments. A logic program is described by a set of rules which defines when a predicate holds. The rules are positive Horn clauses and they are defined using a notation like:

P :- P_1, P_2, ..., P_n.

indicating that the predicate P is true when all conditional predicates P_i are also true. Unconditional rules (with no conditional predicates):

P.

are called *facts* and indicates that the predicate P is true. An example of logic program is showed in Figure 1.
The basic task performed by logic programs is *query* about predicates. A query is used to check if a set of predicates are true according to the program sentences. The algorithm used to test a query is:

Test query: p_1,p_2,...,p_m

1. find a rule: "q :- c_1, ... , c_n", $n \geq 0$. and $p_1[V_1/C_1] = q[V_2/C_2]$, where $q[V_2/C_2]$ means the predicate q with the variables V_2 replaced by the terms C_2.
2. if (found no rule) then output "query is false" ; finish
3. if (found a rule)
   (a) replaces the query to $c_x[V_2/C_2]$, $p_y[V_1$ / $C_1]$
   (b) if (the resulted query is empty) then output "query is true"; finish

(c) goto 1;

Another important property of logic programs is the existence of implicit backtracking. If the used rule does not satisfy the query, the older state of the query is retrieved and another applicable rules are tried until there is no more possible rules.

3 Semantics for Logic Languages

To express accurately the semantic of a logic programming language, the researcher needs to specify:

- the process of searching for applicable rules;
- the variable substitution process;
- the retrieve of old states when backtracking;
- etc.

Some researchers [NF89] tried to give the semantics of PROLOG (the most known logic programming language) using the environments given by common formal methods like denotational semantics. The results are too complex and hard to be understood and reused.

4 Logic Extension of Action Notation

To simplify the description of logic programming languages we propose the definition of new actions to describe the basic concepts found in this kind of language. These new actions can be divided in two classes:

- *control actions* to model the control structures existing in logic programming languages; and
- *unification actions* to model the variables existing in logic programming languages.

4.1 Control Actions

We define the following control actions:

(1) _ backtracking _ :: action, action → action .
(2) continue : action .
(3) terminating _ :: action → action .
(4) terminate : action .

The action combinator (_ backtracking _) was designed to model the sequencing command existing in the logic paradigm. When the action a backtracking b is performed the action a is performed. During the performance of a, the performing of the action continue transfers the control to the second action (b) of the last performed backtracking combinator. After the performance of b the control returns to after the performed continue action. To exemplify the use of these actions, when the action:

```
(1) |give 0
    |then
(2) |continue
    |then
(3) |continue
    backtracking
(4) |give the successor of the given integer
```

is performed, the action (1) produces a transient information which is passed to the continue action (2), this action transfers the execution to the action (4). When (4) is performed the execution returns to action (3) (the action following (2)) that will perform (4) again. After the performance of (4), the control returns to the action following action (3), which does not exist, then the whole action terminates. The transients given to the action (4) are the same ones given to the caller actions (2) and (3). The bindings received by the action (4) are the bindings received be the backtracking combinator.
These actions are useful to describe logic programs. In fact, PROLOG rules like: "P1 :- P2 , P3", can be directly expressed using the action: "P2 backtracking P3"; and PROLOG facts like: "P1.", can be expressing using the action continue. The semantics of the example showed in Figure 1 (not considering the semantics of variables) can be expressed by the following program action:

```
(1) bind "parent" to the closure abstraction
(2) ||give ( "Peter", "John") then continue
    and then
(3) ||give ( "Wally", "Fred") then continue
    and then
(4) ||give ( "John", "Al") then continue
    hence
(5) ||enact the abstraction bound to "parent"
(6) |backtracking
(7) ||enact the abstraction bound to "parent"
(8) |backtracking
(9) ||... do something ...
```

The action (1) creates a binding between the token "parent" and the abstraction which represents the predicate "parent". The semantics of this abstraction is try to continue the program for each possible solution to the query (lines (2),(3) and (4)). The query is modeled by the action in the lines (5)-(7). This action calls the predicate parent two times combining with _ backtracking _ and make some undefined task in the (8). The action (8) will be performed for all solutions found by the both performances of "parent".
Another set of control actions, defined to handle the control transfers existing in logic programming languages, are formed by the actions terminate and terminating a. The action terminating a performs the action a which can perform some backtracking actions. During the performance of a the action terminate aborts all backtracks performed inside a and transfers the execution point to after the terminating action. For example, when the action:

```
(1) |continue and then continue
    backtracking
(2) |terminating ( continue and then continue)
    backtracking
(3) |terminate
```

is performed, the continue actions in (1) perform the actions in (2) twice. When (2) is performed it should perform (3) twice too, but (3)

performs the action terminate which cancels the performance of (3) and the backtracking of (2) and returns immediately to (1). A similar action like:

```
(1) |terminating (continue and then continue)
    backtracking
(2) |continue and then continue
    backtracking
(3) |terminate
```

performs differently because when (3) is performed, both backtracking actions are aborted.

4.2 Unification Actions

Unification actions define actions to express the variable behavior in logic programming languages. The defined actions were designed to handle unifiable cells. An unifiable cell is a memory location which can store the following values:

- the undefined value, meaning that there is no value which unifies with the cell;
- an unifiable value, indicating that this value is unified with the cell;
- another unifiable cell, indicating that the cell is unified with the cell it points to.

The values stored in unifiable cells are changed by unifications. The unification process has the following alternatives:

- When an unifiable cell is unified with a value, the value [stored in] unified with this unifiable cell is compared with the given value. If both values are the same, the unification succeeds. If these values are different, the unification fails. If the unifiable cell is not already unified with any value, the unification succeeds too.
- When two unifiable cells are unified, the values unified with these cells are compared and the unification succeeds if the values are the same. If one of them is not unified with any value, the value stored by this cell is changed to the value stored by the other cell; If both cells stores the undefined value, the unifiable binds the values stored by the cells and next changes in one of these cells will affect the other cell too;
- When two values are unified, the both values are compared. A definition for the comparation of two values is language dependent and is left opened to be defined by the language definition.

The defined actions designed to handle unifiable cells are:

- the action allocate an unifiable cell reserves an unused unifiable cell. Initially, the returned cell is not unified with any value;
- the action unify x with y unifies the values produced by the yielders x and y;
- the action unify values x and y is called by the action unify x with y to compare two values, the semantics of this action should be defined by the user during the language specification. The semantics of this action should perform the action unification error if the unification can not be done;

– the action unifying a, performs the action a and when its performance finishes, the state of the unifying memory is recovered, erasing all unifications performed by a. This action also captures escapes produced by wrong unifications made in a;
– the yielder the x unified with y produces the value stored in the position y.

The unification actions use the following data structures:

– the sort unifiable represents values which can be stored in unifiable cells. Its definition is left open and should be defined by languages specifications;
– the sort unifiable-cell represents memory positions in the unifiable memory. It was defined like ordinary cell positions;

Some examples of actions using the unification are shown below:

```
(1) ||allocate an unified cell
    |then
    ||bind "i" to the given unified-cell
    hence
(2) ||unify the unifiable-cell bound to "i" with 4
    |and then
(3) ||unify the unifiable-cell bound to "i" with 5.
```

The action (1) produces a binding between a token and a new unifiable memory position. The action (2) unifies this position with the value 4. After this, the action (3) tries to unify the same position with the value 5. This action will not be succeed because it is inconsistent with the last unification.

Consider the following action:

```
(1) |||allocate an unified cell
    ||then
    |||bind "i" to the given unified-cell
    |and
(2) |||allocate an unified cell
    ||then
    |||bind "j" to the given unified-cell
    hence
(3) ||unify the unified-cell bound to "i" with
    ||the unified-cell bound to "j"
    |and then
(4) ||unify the unified-cell bound to "i" with 5
    |and then
(5) ||give the number unified with the unified-cell bound to "j"
```

Actions (1) and (2) make bindings between tokens and new unifiable memory positions. The action (3) unifies these memory positions. The action (4) unifies the first memory position with the value 5. It makes with the second memory position be unified with these value too, because both memory position are unified. The action (5) tests this fact returning the value unified with the second memory position.

In the action

```
(1) ||allocate an unified cell
    then
    ||bind  "i" to the given unified-cell
    hence
(2) ||unifying
    |||unify the unified-cell bound to  "i" with 5
    and then
(3) ||unify the unified-cell bound to  "i" with 4.
```

the action (1) produces a binding between a token and a new unifiable memory position. The action (2) makes an unification inside the combinator unifying, this unification will be erased when this action completes. The action (3) tests this property by making another unification that should be inconsistent with that made in (2).

These proposed actions are useful to model the behavior of the variables in logic programs. The full program action that gives the semantics of the program showed in Figure 1 is:

```
(1) |bind  "parent" to the closure abstraction
(2) ||unifying
    |||unify the given datum#1 with  "Peter"
    |||and then
    |||unify the given datum#2 with  "John"
    |||and then continue
    ||and then
(3) ||unifying
    |||unify the given datum#1 with  "Wally"
    |||and then
    |||unify the given datum#2 with  "Fred"
    |||and then continue
    ||and then
(4) ||unifying
    |||unify the given datum#1 with  "John"
    |||and then
    |||unify the given datum#2 with  "Al"
    |||and then continue
    |hence
(5) ||allocate an unified cell then bind  "X" to it
    ||and
    ||allocate an unified cell then bind  "Y" to it
    ||and
    ||allocate an unified cell then bind  "Z" to it
    |before
(6) |||give (the value bound to  "X",the value bound to  "Y")
    ||then
    ||enact the abstraction bound to  "parent"
    |backtracking
(7) |||give (the value bound to  "Y",the value bound to  "Z")
    ||then
    ||enact the abstraction bound to  "parent"
    |backtracking
(8) ||ShowAnswer.
```

The action `(1)` produces the bindings for the predicate, this abstraction tries each possible solution (actions `(2)`,`(3)` and `(4)`). For each solution, the action unifies the given values with the current solution and continues the execution. The action `(5)` produces the bindings for the used variables, these variables are used in actions `(6)` and `(7)` when enacting the queries. Finally, the action `(8)` is performed for each possible solution found and should make some processing to use it.

5 Case Study: PROLOG Action Semantics

To test the semantic entities proposed in this paper we describe the PROLOG programming language.

5.1 Abstract Syntax

Program = ⟦ Predicate$^+$ "in" Variable Question$^+$ ⟧.
Rule = ⟦ "pred" identifier "is" Variable Clause$^+$ "end" ⟧.
Question = ⟦ ":-" Predicate$^+$ "." ⟧.
Variable = ⟦ "vars" Identifier$^+$ "." ⟧
Clause = ⟦ Predicate ":-" Predicate$^+$ ⟧ |
⟦ Predicate "." ⟧.
Predicate = ⟦ "!" ⟧ | ⟦ Identifier "(" Term$^+$ ")" ⟧.
Term = Variable | Constant | ⟦ Identifier "(" Term$^+$ ")" ⟧.

5.2 Semantic Functions

- run _ :: Program → action.

(1) run ⟦ p: (Predicate$^+$) "in" v:Variable q :Question$^+$ ⟧ =
||predicate elaborate p
and
||variable elaborate v
hence
|execute q.

- variable elaborate _ :: Variable → action.

variable elaborate ⟦ "vars" (i_1,i_2) "." ⟧ =
|variable elaborate ⟦ "vars" i_1 "." ⟧
and
|variable elaborate ⟦ "vars" i_2 "." ⟧

variable elaborate ⟦ "vars" i:Identifier "." ⟧ =
|allocate an unifiable cell
then
|bind i to the given unifiable-cell

(1) predicate elaborate _ :: Predicate$^+$ → action.
predicate elaborate () = complete.
predicate elaborate (p_1,p_2) = elaborate p_1 and elaborate p_1.
predicate elaborate ⟦ "pred" i:Identifier "is" v:Variable c:Clause$^+$
"end" ⟧ =
|bind i to abstraction of
|||furthermore variable elaborate v
|hence
|||try c.

- execute _ :: Question$^+$ → action.

execute (q_1,q_2) =
|execute q_1
and then
|execute q_2.

execute ⟦ ":-" p:Predicate$^+$ "." ⟧ =
|do p
backtracking
|complete.

- try _ :: Clause$^+$ → action.

try (c_1,c_2) = try c_1 and then try c_2.
try ⟦ p_1:Predicate "`:-`" p_2:Predicate$^+$ ⟧ =
unifying
|check predicate p_1
|and then
||do p_2
|and then
||continue.
try ⟦ p:Predicate "`.`" ⟧ =
unifying
|check predicate p_1
and then
|continue.

- check predicate _ :: Predicate → action.
check predicate ⟦ Identifier "`(`" t:Term$^+$ "`)`" ⟧ =
|||evaluate t
||then
|||give operator "`()`" with the given data
|and
||give operator "`()`" with the given data
then
|unify the given term#1 with the given term#2.

- do _ :: Predicate$^+$ → action.
do (p_1,p_2) =
|do p_1
backtracking
|do p_2.
do ⟦ "`!`" ⟧ =
|continue
and then
|terminate.
do ⟦ p:Identifier "`(`" t:Term$^+$ "`)`" ⟧ =
|evaluate t
then
|enact the application of
|(the abstraction bound to p)
|to the given data.

- evaluate _ :: Term$^+$ → action.
evaluate (t_1,t_2) = evaluate t_1 and then evaluate t_2.
evaluate v:Variable = give the value bound to v.
evaluate c:Constant = give the value of c.
evaluate ⟦ i "`(`" t "`)`" ⟧ =
|evaluate t
then
|give operator i with the given data.

5.3 Semantic Entities

introduces: term, operator _ with _, the operator of _, arguments of _.

- (operator t_1 with a_1) is compatible with (operator t_2 with a_2) =
  both(t_1 is t_2, count a_1 is count a_2).

- unifiable = string | token | number | term.

  unify value t_1:term t_2:term =
  ||bind "T1" to t_1
  and
  ||bind "T2" to t_2
  hence
  |||check the term bound to "T1" is
  |||compatible with the term bound to "T2"
  |and then
  |||give 1
  |then
  |||unfolding
  ||||unify the (the given number) argument of
  ||||(the term bound to "T1") with
  ||||the (the given number) argument of
  ||||(the term bound to "T2")
  |and then
  ||||check not the given number is
  ||||count of arguments of
  ||||(the term bound to "T1")
  |and then
  |||give the successor of the given number
  |then
  |||unfold
  |or
  ||||check the given number is
  ||||count of arguments of
  ||||(the term bound to "T1")
  |and then
  |||complete
  or
  |||check not the term bound to "T1"
  |is compatible with the term bound to "T2"
  |and then
  |||escape with unification-error.

- v_1 : (string | token | number) ⇒
  unify values v_1 v_2 =
  |give v_1 is v_2
  then
  ||complete
  else
  ||escape with unification-error.

```
    v2 : (string | token | number) ⇒
        unify values v1 v2 =
        |give v1 is v2
        then
        ||complete
        |else
        ||escape with unification-error.
```

- term = operator token with data.
- the operator of (operator t with a) = t.
- arguments of (operator t with a) = a.
- the n argument of (operator t with a) = a.

6 Conclusions and Future Works

We showed that action semantics can be used to express the semantics of logic programming languages like PROLOG. The semantic entities proposed in this paper can be easily described using standard action notation as we show in Appendix A. But our experience shows that they are more powerful to describe logic programming languages, because it makes specifications more legible and simpler than if we use only the standard action notation. These facts can justify the adoption of the entities showed in this paper for action semantic descriptions of programming languages.
Thus, using the proposed semantic entities, we think that research and application on semantics of this kind of languages and the design of multi-paradigm languages becomes simpler.

References

[NF89] T. Nicholson and N. Foo. A denotational semantics for prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, oct 1989.

A Formal Specification for the Logic Actions

The next sections will define the semantics for the proposed action combinators using the original action notation.

A.1 Control Actions

The semantics of the backtracking combinators can be expressed by the following specification:

- action-backtracking : token.
- next-action : cell.
- _ backtracking _ :: action, action → action.
- _ backtracking _ :: action, action → action.

(1) a backtracking b =

```
|store (closure abstraction of b,the data stored in next-action)
|in next-action
and then
|||a
||and then
|||store the rest of the data stored in next-action
|||in next-action
|trap
|||store the rest of the data stored in next-action
|||in next-action
||and then
|||escape with them
```

- continue : action.

(2) continue =

```
|give the data stored in next-action
then
||store the rest of them in next-action
and then
||enact the abstraction#1
and then
||store them in next-action.
```

A.2 Unification Action

The semantics of the unification actions can be expressed by the following specification:

A.2.1 Storage

- unifiable-cell = cell [unifiable | unifiable-cell | undefined ⟧ .
- undefined : datum .
- unifiable = □ .

An unifiable-cell is a cell which can stored the value undefined, an unifiable or another unifiable-cell. The sort unifiable defines the values which can be stored on unifiable cells and it should be defined by programming language specifications. The idividual undefined is a datum used to specify that there is no value stored in some unifiable cell.

- allocate an unifiable cell =

```
|allocate a cell
then
||store undefined in it
and
||regive.
```

allocating a new unifiable cell has the same semantics for allocate a ordinary cell, the initial value of an unifiable cell is undefined.

- last cell _ :: unifiable-cell → yielder[producing an unifiable-cell].
  c_2 = the datum stored in c ⇒
  last cell c_1 = if (c_2 : datum) then last cell c_2 else c_1.
- the _ unificated with _ :: unifiable, unifiable-cell → yielder.

the d unificated with c = the d stored in last cell c.

- restore _ to _ and _ :: unifiable-cell, unifiable, restore → restore.
- stop _ :: restore → restore.
- no-restore : restore.
- changes-cell : cell[restore].
- undo-changes _ :: restore → action.
  undo-changes no-restore = give no-restore.
  undo-changes stop r = give r.
  undo-changes (retore c to u and r) =
  |store u in c
  and then
  |undo-changes r.
  undo : action.
  undo =
  |undo-changes the changes stores in changes-cell
  then
  |store the given restore in changes-cell.
- unifying _ :: action → action.
  unifying a =
  ||store (stop the restore stored in changes-cell) in changes-cell
  and then
  |||a
  and then
  ||undo
  trap
  |||check the given data is unification-failing
  and then
  ||undo
  or
  |||check not the given data is unification-failing
  and then
  ||undo
  and then
  ||escape with the given data

- unify _ with _ :: yielder, yielder → action.
  unify d_1 with d_2 =
  |give last-cell d_1 and give last-cell d_2
  then
  |do unify the given data
- undoable store _ in _ :: cell, storable → action.
  undoable store v in c =
  |store (restore (the value stored in c) to c and
  |(the changes stored in changes-cell)) in changes-cell
  and then
  |store v in c.
- do unify _ :: (data, data) → action.
  do unify (v_1:unifiable, v_2:unifiable) = unify values v_1 and v_2.

```
do unify (c1:unifiable-cell, c2:unifiable)=
    |give the value stored in c1 is undefined
    then
    ||undoable store c2 in c1
    |else
    ||do unify (the value unified with c1, c2)
do unify (c1:unifiable, c2:unifiable-cell)=
    |give the value stored in c2 is undefined
    then
    ||undoable store c1 in c2
    |else
    ||do unify (the value unified with c1, c2)
do unify (c1:unifiable-cell, c2:unifiable-cell)=
    |give the value stored in c1 is undefined
    then
    ||undoable store c2 in c1
    |else
    ||do unify (the value unified with c1, c2)
```

CASL and Action Semantics

Peter D. Mosses

BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark

Abstract. CASL, the Common Algebraic Specification Language, might be used as a meta-notation in action-semantic descriptions, instead of Unified Algebras. However, it appears that direct use of CASL as a meta-notation would have some drawbacks; a compromise is proposed.

1 Background

CASL, the Common Algebraic Specification Language [2, 9, 10] developed by CoFI, the Common Framework Initiative [1], is an expressive language intended for use when specifying software requirements and design. Basic specifications in CASL allow declaration of both total and partial operations, predicates, subsorts, and datatypes with constructors and (optionally) selectors; axioms are first-order formulae, and mixfix notation may be used. Structured specifications allow extension, translation, hiding, both loose and initial semantics, and generic specifications. A higher-order extension of CASL [5] allows operations and predicates to be passed as arguments, and provides notation for tuples.

Several libraries of basic datatypes in CASL have been developed [13]. Moreover, the use of CASL is supported by the availability of parsers and interfaces to theorem-provers [4]. The formal semantics of CASL has been defined [3].

A simple (yet expressive) framework called Unified Algebras (UA) [7] has previously been used as meta-notation in Action Semantics for defining Action Notation and Data Notation, and for specifying abstract syntax, semantic functions and semantic entities in action-semantic descriptions [8]. Unfortunately, UA is not widely known, and is generally regarded as somewhat idiosyncratic. Using CASL instead of UA would significantly increase the accessibility of Action Semantics. Moreover, the use of sorts as values in UA could be represented in CASL by declaring constants corresponding to the sorts, and by providing operations such as union and intersection on these sort representatives. The basic datatypes of CASL could replace the standard Data Notation [8, App. E]. (The higher-order extension of CASL is needed for specifying the embedding of data operations in Action Notation, and for providing notation for tuples, but the specification of data may itself remain first-order.)

In this paper, we first give an overview of CASL, indicating how its various constructs are written. We then consider the use of CASL for specifying Data Notation, Action Notation, abstract syntax of programming languages, semantic functions, and semantic entities. We also discuss the use of CASL to express the modular structure of action-semantic descriptions. Our tentative conclusion

is that it could be advantageous to use CASL for specifying Data Notation, Action Notation, and semantic functions, hereas there appear to be significant drawbacks to using it for specifying abstract syntax and semantic entities, and for expressing the modular structure of action-semantic descriptions..

The reader is assumed to be familiar with Action Semantics, and with the use of Unified Algebras in action-semantic descriptions [8, Ch. 3 and App. F].

2 Introduction to CASL

2.1 Overview

CASL, the Common Algebraic Specification Language [2, 9, 10], has been developed by CoFI, the Common Framework Initiative [1]. Its main features are as follows:

- The CASL design is based on a critical selection of the concepts and constructs found in existing frameworks.
- CASL is an expressive specification language with simple semantics and good pragmatics.
- CASL is appropriate for specifying requirements and design of conventional software packages.
- There is a coherent family of sub-languages and extensions of CASL.

CASL consists of the following major parts, which are quite independent and may be understood (and used) separately:

Basic Specifications

- Basic specifications denote classes of partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed.
- Subsorts are interpreted as embeddings.
- Axioms are first-order formulae built from definedness assertions and both strong and existential equations. Sort generation constraints can be stated.
- Datatype declarations are provided for concise specification of sorts equipped with some constructors and (optional) selectors, including enumerations and products.

Structured Specifications

- Structured specifications allow translation, reduction, union, and extension of specifications.
- Extensions may be required to be free; initiality constraints are a special case of free extensions.
- A simple form of generic (parameterized) specification is provided, together with instantiation involving parameter-fitting translations.

Architectural Specifications

– Architectural specifications express that the specified software is to be composed from separately-developed, reusable units with clear interfaces.

Libraries

– Libraries allow the distributed storage and retrieval of named specifications.

The following sections indicate how the various constructs of CASL are written.

2.2 Basic Specifications

sorts ... **ops** ... **preds** ... **types** ... **vars** ... **axioms** ...
: lists symbol declarations/definitions and axioms, which may be given in any order with declaration occurring before use.

keyword item1;... *itemn*;
: abbreviates *keyword item1*;... *keyword itemn*; the terminating semicolon is optional, as is the use of the plural form of *keyword*.

Symbol Declarations and Definitions

Sorts (s)

sorts $s1, \ldots, sn$
: declares some sorts.

sorts $s1, \ldots, sn < s$
: declares some subsorts and their supersort.

sort $s = \{v : s' \bullet F\}$
: declares the sort s to consist of those values of the variable v in s' for which the formula F holds.

Operations (f)

op $f : s1 \times \ldots \times sn \to s$
: declares a total function with argument sorts $s1$, ..., sn and result sort s.

op $f : s1 \times \ldots \times sn \to ? s$
: declares a partial function with argument sorts $s1$, ..., sn and result sort s.

op $f : \ldots, assoc$
: declares a binary function to be associative.

op $f : \ldots, comm$
: declares a binary function to be associative.

op $f : \ldots, idem$
: declares a binary function to be idempotent.

op $f : \ldots, \mathit{unit}\ T$
: declares a binary function to have the value of the term T as both left and right units.

ops $f1, \ldots, fn : \mathit{type}$
: abbreviates **op** $f1 : \mathit{type}; \ldots;$**op** $fn : \mathit{type}$.

op $f(v1 : s1; \ldots; vn : sn) : s = T$
: declares a total function with argument sorts $s1, \ldots, sn$ and result sort s, and specifies its value on $v1, \ldots, vn$ to be the value of the term T.

op $f(v1 : s1; \ldots; vn : sn) :?s = T$
: declares a partial function with argument sorts $s1, \ldots, sn$ and result sort s, and specifies its value on $v1, \ldots, vn$ to be the (perhaps undefined) value of the term T.

Constants (c)

op $c : s$
: declares a constant whose value is of sort s.

op $c :?s$
: declares a constant whose value is of sort s or undefined.

op $c : s = T$
: declares a constant of sort s and specifies its value to be the value of the term T.

op $c :?s = T$
: declares a constant whose value is of sort s and specifies its value to be the (perhaps undefined) value of the term T.

Predicates (p)

pred $p : s1 \times \ldots \times sn$
: declares a predicate with argument sorts $s1, \ldots, sn$.

pred $p(v1 : s1; \ldots; vn : sn) \Leftrightarrow F$
: declares a predicate with argument sorts $s1, \ldots, sn$ and specifies its holding on $v1, \ldots, vn$ to be the same as that of the formula F.

preds $p1, \ldots, pn : \mathit{type}$
: abbreviates **pred** $p1 : \mathit{type}; \ldots;$**pred** $pn : \mathit{type}$.

Datatypes

types $s1 ::= A1; \ldots; sn ::= An$
: declares the sorts $s1, \ldots, sn$ to have the constructors, selectors, and subsorts specified by the corresponding alternatives $A1, \ldots, An$, which may use sorts before they are declared.

generated types $s1 ::= A1; \ldots; sn ::= An$
: declares also that the sorts $s1, \ldots, sn$ are generated by their constructors.

free types $s1 ::= A1; \ldots; sn ::= An$
: declares also that the sorts $s1, \ldots, sn$ are uniquely generated by their constructors.

Alternatives (A)

$f(s1; \ldots; sn)$
: declares f to be a total constructor function with argument sorts $s1$, $\ldots$, sn.

$f(s1; \ldots; sn)?$
: declares f to be a partial constructor function with argument sorts $s1$, $\ldots$, sn.

c
: declares c to be a constant constructor value.

$f(\ldots; f1, \ldots, fm : si; \ldots) \ldots$
: declares f to be a constructor operation with m arguments of sort si, and $f1$, $\ldots$, fm to be total selector operations with result sort si.

$f(\ldots; f1, \ldots, fm :? si; \ldots) \ldots$
: declares f to be a constructor operation with m arguments of sort si, and $f1$, $\ldots$, fm to be partial selector operations with result sort si.

sorts $s1', \ldots, sk'$
: declares the embeddings of the sorts $s1'$, $\ldots$, sk' to be total constructor operations.

$A1 \mid \ldots \mid An$
: lists multiple alternatives.

Sort Generation

generated { sorts $\ldots$ **ops** $\ldots$ **preds** $\ldots$ **types** $\ldots$ **}**
: declares that the sorts declared by the grouped symbol declarations and definitions are generated by the operations declared there.

Axioms

axioms $F1; \ldots; Fm$
: asserts that all the formulae $F1$, $\ldots$, Fm hold for all values of the globally-declared variables.

vars $v1, \ldots, vn : s$
: declares the variables $v1$, $\ldots$, vn for use in subsequent axioms.

vars $v1, \ldots, vn : s \bullet F1 \ldots \bullet Fm$
: asserts that all the the formulae $F1$, $\ldots$, Fm hold for all values of the variables $v1$, $\ldots$, vn.

Formulae (F)

$\forall\, v1, \ldots, vn : s \bullet F$
: is universal quantification of F with the variables $v1$, ..., vn.

$\exists\, v1, \ldots, vn : s \bullet F$
: is existential quantification of F with the variables $v1$, ..., vn.

$\exists! v1, \ldots, vn : s \bullet F$
: is unique-existential quantification of F with the variables $v1$, ..., vn.

$\forall \ldots ; \ldots \bullet F$
: abbreviates $\forall \ldots \bullet \forall \ldots \bullet F$, and similarly for existential quantification.

$F1 \wedge \ldots \wedge Fn$
: is conjunction of formulae.

$F1 \vee \ldots \vee Fn$
: is disjunction of formulae.

$F \Rightarrow F'$
: is implication.

$F'\ if\ F$
: is reverse implication.

$F \Leftrightarrow F'$
: is equivalence.

$\neg\, F$
: is negation.

$true, false$
: are constant formulae.

$p(T1, \ldots, Tn)$
: is application of a predicate p to argument terms.

$t0\ \ T1\ \ t1 \ldots Tn\ \ tn$
: is mixfix application of a predicate $t0__t1 \ldots __tn$ to argument terms.

q
: is use of a constant predicate q.

$T = T'$
: is ordinary (strong) equality, holding also when the values of both T and T' are undefined.

$T \stackrel{e}{=} T'$
: is existential equality, holding only when the values of both T and T' are defined.

$T \in s$
: is subsort membership, holding when the value of T is in the subsort s.

Terms (T)

$f(T1, \ldots, Tn)$
: is application of a function f to argument terms.

t0 T1 t1 ... Tn tn
: is mixfix application of a function $t0__t1 \ldots __tn$ to argument terms.

t0 T1, ..., Tn t1
: is literal syntax for repeated application of a binary function (associated with $t0__t1$ by a list annotation) to argument terms.

c
: is use of a constant value c.

$T : s$
: is interpreting T as a value of sort s.

T as s
: is projecting T onto a subsort s.

T when F else T′
: is conditional choice between T and T', depending on whether the formula F holds or not.

Symbols (*SY*)

Words are sequences formed from:

$A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, ', _$,

starting with a letter (optionally preceded by a dot), with no double underscores, and different from the *reserved words*:

and, arch, as, assoc, axiom, axioms, closed, comm, def, else, end, exists, false, fit, forall, free, from, generated, get, given, hide, idem, if, in, lambda, library, local, not, op, ops, pred, preds, result, reveal, sort, sorts, spec, then, to, true, type, types, unit, units, var, vars, version, view, when, with, within.

Signs are sequences formed from:

+, −, ∗, /, \, &, =, <, >, !, ?, :, ., $, @, ˆ, ˜, |, [,], {, },
¡, ¿, ×, ÷, £, ©, ±, ¶, §, ^{1}, ^{2}, ^{3}, ·, ∉, °, ¬, μ

different from the reserved signs:

:, :?, ::=, =, =>, <=>, ., ·, |, |−>, \/, /\, ¬.

Mixfix symbols sre sequences formed from *single* words or signs separated by *double* underscores __ as place-holders, and with any brackets ([,], {, }) being balanced.

Numbers are of the form:

n, $n.n'$, nEn', $n.n'En''$,

where n, n', n'' are sequences of decimal digits.

Characters are of the form `'c'`, where c is a single character or an escape sequence starting with \.

Strings are of the form `"`$c1 \ldots cn$`"`, where each ci is a single character or an escape sequence starting with \.

2.3 Structured Specifications

Specifications (*SP*)

SP **with** *SM*
: translates the symbols declared by *SP* using the symbol map *SM*.

SP **hide** *SL*
: hides the symbols declared by *SP* that are also in the symbol list *SL*.

SP **reveal** *SM*
: hides the symbols declared by *SP* other than those listed or mapped by the symbol map *SM*.

SP1 **and** ... **and** *SPn*
: is union of specifications.

SP1 **then** ... **then** *SPn*
: is extension of specifications.

free *SP*
: is free extension when used in an extension, and restriction to initial models otherwise.

local *SP* **within** *SP'*
: is local specification of hidden symbols.

closed *SP*
: is ensuring that *SP* is not interpreted as an extension.

Named and Parameterized Specifications

spec *SN* = *SP* **end**
: is naming a specification (the **end** is optional).

spec *SN*[*SP1*]...[*SPn*] = *SP* **end**
: is naming a specification with parameters *SP1*, ..., *SPn*.

spec *SN*[*SP1*]...[*SPn*] **given** *SP1''*,...,*SP1''* = *SP* **end**
: is naming a specification with parameters *SP1*, ..., *SPn* and imports *SP1''*, ..., *SP1''*.

SN
: is reference to the non-parameterized specification named *SN*.

SN[*FA1*]...[*FAn*]
: is instantiation of the parameterized specification named *SN* with fitting arguments *FA1*, ..., *FAn*.

Fitting Arguments (FA)

SP′ **fit** *SM*
: is fitting the symbols declared by the parameter specification to those declared by the argument *SP′* using the symbol map *SM*.

SP′
: is fitting the symbols declared by the parameter specification to those declared by the argument *SP′* using a uniquely-determined implicit symbol map.

FV
: is fitting the symbols declared by the parameter specification using a fitting view *FV*.

Named and Parameterized Views

view *VN* : *SP* **to** *SP′* = *SM* **end**
: is naming the view from *SP* **to** *SP′* determined by the symbol map *SM* (the **end** is optional).

view *VN*[*SP1*]...[*SPn*] : *SP* **to** *SP′* = *SM* **end**
: is naming a view with parameters *SP1*, ..., *SPn*.

view *VN*[*SP1*]...[*SPn*] **given** *SP1″*,...,*SP1″* : *SP* **to** *SP′* = *SM* **end**
: is naming a view with parameters *SP1*, ..., *SPn* and imports *SP1″*, ..., *SP1″*.

Fitting Views (FV)

view *VN*
: is reference to the non-parameterized view named *VN*.

view *VN*[*FA1*]...[*FAn*]
: is instantiation of the parameterized view named *VN* with fitting arguments *FA1*, ..., *FAn*.

Symbol Lists (*SL*) and Maps (*SM*)

SY1, ..., *SYn*
: lists the symbols, optionally with keywords (**sorts**, **ops**, and **preds**) to indicate the kind of the subsequent symbols in the list.

SY1 ↦ *SY1′*, ..., *SYn* ↦ *SYn′*
: maps each symbol *SYi* to *SYi′*, optionally with keywords (**sorts**, **ops**, and **preds**) to indicate the kind of the subsequent symbols in the map; identity maps *SYi* ↦ *SYi′* may be abbreviated to *SYi*, so a symbol list *SL* is a special case of a symbol map.

2.4 Architectural Specifications

Omitted here, since the structure of models of specifications appears to be irrelevant to Action Semantics.

2.5 Specification Libraries

library *LN* ...
: associates a library name *LN* with a sequence of downloadings, named specifications, and/or views.

Downloadings

from *LN* **get** *IN1*,..., *INn* **end**
: copies the items with the listed names *IN1*,..., *INn* from the library named *LN*.

from *LN* **get** ... $IN \mapsto IN'$ ... **end**
: renames the copied item named *IN* to IN'.

Library Names (LN)

LI **version** *N1*. ... .*Nm*
: refers to a particular version of the library with identifier *LI*.

FI1/.../*FIn*
: identifies an installed library; an uninstalled library is identified by its (absolute) URL.

2.6 Comments and Annotations

Comments are of the form:

%{ ... **}%** or %% ...

the latter being terminated by the end of the line.

Annotations are generally of the form:

%word(...**)%** or **%word** ...

the latter being terminated by the end of the line. Some annotations may affect parsing and display throughout the enclosing library; others record purported facts about the specification (e.g. that an extension is conservative). An annotation of the form:

%(...)%

generally labels the preceding item with '...'.

3 Specifying Data Notation

CASL might be considered as an attractive language for specifying the abstract datatypes that form the Data Notation used in Action Semantics: the libraries of basic datatypes being developed for CASL are already more comprehensive than the standard Data Notation [8, App. D]; tools for checking CASL specifications have been implemented; CASL interfaces to interactive theorem-provers (such as HOL/Isabelle and INKA) are available; and CASL is already quite widely known (at least within the algebraic specification community).

However, there could be some drawbacks. For instance, the symbols for operations and predicates declared by the CASL libraries of basic datatypes tend to be conventional mathematical signs (such as + and $\leq$), in marked contrast to the suggestive words used in the standard Data Notation (e.g. *sum* and *_is less than_*). A translated version of each CASL library, declaring verbose symbols rather than mathematical signs, could be provided (perhaps using the CASL construct for translation rather than actually carrying out the translation); those familiar with the CASL libraries from their use in other contexts might however find such a translation counter-productive and alienating. On the other hand, it is conceivable that users of Action Semantics might welcome the more conventional mathematical notation for familiar basic datatypes provided by the CASL libraries, despite the lack of backwards compatibility with previous action-semantic descriptions.

Apart from the style of symbols used for data operations and predicates in standard libraries, there are are some general differences between what is allowed in CASL and the Unified Algebras meta-notation previously used in Action Semantics. For instance, at the lexical level, place-holders in mixfix operation and predicate symbols are written as *double* underscores __ in CASL, and single underscores are used to separate words—spaces and hyphens are not allowed in verbose symbols. Thus the UA symbol *_is in_* would have to be written *__is_in__* when declared in CASL. A benefit of the CASL use of single underscores instead of spaces is that the intended grouping of words into complete operation symbols may be clearer than in UA specifications. However, terms and formulae in CASL cannot mimic natural language as closely as those in UA.

A minor lexical bother is that the words *true* and *false* in CASL are reserved for atomic formulae, and cannot be used as Boolean constants; an alternative spelling would be needed. More seriously, commas and parentheses are reserved signs in CASL, and cannot be used at all in declared symbols, so the notation for tuples provided by the standard Data Notation (and used extensively throughout action-semantic descriptions) cannot be specified directly. A higher-order extension of CASL provides what looks like the desired notation for tuples, e.g. allowing $(1, 2, 3)$; however, tuple construction there is non-associative, so that $(1, (2, 3))$, $((1, 2), 3)$, and $(1, 2, 3)$ are all different, in contrast with their interpretation in the standard Data Notation. Thus use of CASL would require either abandoning the associativity of tupling, or the use of a different notation (such as $\langle 1, 2, 3 \rangle$) for associative tuple construction.

CASL does not allow the declaration of sort-constructing operations (not even in its higher-order extension). It would however be quite straightforward to represent sorts by ordinary values, equipping these values with membership and inclusion predicates, and with union and intersection operations; then operations representing sort constructors may be specified much as in UA (for instance the construction of sorts of lists from sorts of components).

CASL allows the use of compound identifiers for sorts declared in generic specifications, e.g. $List[Item]$ for the sort of lists with components in the parameter sort $Item$. When the generic specification declaring $List[Item]$ is instantiated with Int for $Item$, the resulting sort of lists is written $List[Int]$; however, subsorts such as $List[Nat]$ are not automatically declared, and would have to be specified by separate instantiations. Moreover, the subsort embedding $Nat < Int$ does not entail $List[Nat] < List[Int]$, and such embeddings might be needed.

Apart from the problems concerning notation for tuples, it seems atractive to adopt CASL for specifying Data Notation.

4 Specifying Action Notation

The specification of Action Notation consists of:

- a (possibly modular) structural operational semantics for a kernel of Action Notation;
- a definition of an appropriate equivalence, based on the operational semantics; and
- a reduction from the full Action Notation to the kernel.

CASL is particularly convenient for expressing structural operational semantics: it allows transition relations to be declared as (infix or mixfix) predicates, and specified as the least predicates satisfying some conditional formulae; it also allows the use of partial operations in side-conditions (which is quite common in ordinary SOS, and essential for Modular SOS). For an example, see the Modular SOS description of the original Action Notation [11], which is specified entirely in CASL (except for a rule involving arbitrary data operations, which has to be understood schematically, as in the original definition using UA). Conditional axioms in CASL can be made to resemble the inference rule notation of SOS by using a comment line to separate the conditions from the conclusion.

Thanks to the provision of general first-order formulae, CASL may also be used for formally defining bisimulation and testing equivalence. (Such equivalences could only be described informally in the original definition of Action Notation, since UA is restricted to Horn clauses.)

Use of operation definitions in CASL would allow the reduction of the full Action Notation to its kernel to be specified clearly and concisely. Such explicit definitions may be preferable to the algebraic equations that were used in the original definition, and which left both the kernel and the process of reduction to it rather too implicit.

Thus CASL appears to be well-suited for specifying Action Notation. There would however be some minor bother with using the particular symbols of Action Notation that contain reserved words of CASL (*or*, *and*, *then*): such symbols would have to be written differently when declared and used in CASL specifications.

5 Specifying Abstract Syntax

Abstract syntax is specified in UA using algebraic equations between sort constants and sort terms. The latter are formed from individual characters, list constructors (with literal strings being simply lists of characters), and regular expression constructors: sort union, tuple concatenation, and repetition (arbitrary, at least once, or at most once). Using the keyword '**grammar**' before a set of equations in the UA specification indicates the implicit introduction of all the sort constants given by the left-hand sides of the equations (as well as the importation of standard modules providing the notation for characters, strings, and lists).

grammar: Stm = [["if" Exp "then" Stm ⟨"else" Stm⟩$^?$]] | ....
Exp = ...

Specifying abstract syntax in UA exploits all the main features of the metanotation: equations between sorts, sort union, and application of sort constructors to individuals as well as to proper sorts. At the level of structured specification, mutual recursion between the definitions of syntactic sorts may lead to mutual reference between UA modules.

Although UA allows particularly concise specifications of abstract syntax, the use of double square brackets [[...]] in grammars as list constructors, and the need to enclose terminal symbols in quotes, may be regarded as nuisances, especially by casual readers.

In CASL, the most concise way of specifying abstract syntax is to use datatype declarations. For instance, the abstract syntax of if-then-else statements may be specified as follows:

free types $Stm ::= IF_THEN_ELSE__(Exp; Stm; Stm) \mid \ldots;$
$Exp ::= \ldots$

As with grammars in UA, the sorts that occur on the left-hand sides of the BNF-like productions are implicitly declared, and alternatives for the same sort may be combined using ... | .... In most other respects, however, the expressiveness of the CASL specification is quite restricted. In particular, strings and characters cannot be used directly as component sorts, so suggestive terminal symbols have to be incorporated in the symbols for constructors. Mixfix constructors have to be indicated with explicit place-holders, which may look a bit clumsy. Component sorts cannot be replaced by sort terms—except for compound sort symbols such as $List[Exp]$ (the instantiations that give rise to such sorts would need to be given explicitly, before the datatype declarations). It is unclear how best to

specify optional components in CASL: by introducing a further constructor, such as $IF__THEN__(Exp; Stm)$, or by use of a generic datatype for optional values, as in $IF(Exp; Stm; Opt[Stm])$. By the way, *if* is a reserved word in CASL, hence the uppercase spelling above.

Rather than specifying abstract syntax directly in CASL, it would seem to be preferable to specify it in a meta-notation close to grammars in UA (perhaps eliminating the double square brackets, and exploiting systematic case differences to distinguish sorts from ordinary constant values). From such a specification, a (somewhat less concise) CASL specification could be generated automatically.

6 Specifying Semantic Functions

When using UA as meta-notation in action-semantic descriptions, the operation symbols for semantic functions have to be first uced, and then their functionalities specified separately. In CASL, the declaration of an operation symbol is combined with the specification of its argument and result sorts, so it would be written more succinctly, e.g.:

op $execute : Stm \to Action$

In the semantic equations that define the semantic functions, each variable ranging over a syntactic sort is declared "on the fly" in UA, the sort being indicated when the variable is first used in each equation. In CASL, one may use explicit universal quantification to declare variables for use in a semantic equation; alternatively, declarations of variables that are used in several semantic equations may be collected together and specified either globally or locally. The CASL style appears to have the advantage of making the left-hand sides of semantic equations clearer, e.g.:

op $execute__ : Stm \to Action$
vars $E : Expr; S1, S2 : Stm$
- $execute[[IF\ E\ THEN\ S1\ ELSE\ S2]] =$
  $\ldots evaluate\ E \ldots execute\ S1 \ldots execute\ S2$

Another point to note is that in UA, the double square brackets used in the left-hand side of semantic equations are part of the notation for nodes of abstract syntax trees. In CASL, one might specify them to be an identity operation on each syntactic sort:

ops $[[__]](S : Stm) : Stm = S;$
$[[__]](E : Exp) : Exp = E; \ldots$

Alternatively, the square brackets could be incorporated in the symbols for the semantic functions themselves:

op $execute[[_]] : Stm \to Action$
- $execute[[IF\ E\ THEN\ S1\ ELSE\ S2]] =$
  $\ldots evaluate[[E]] \ldots execute[[S1]] \ldots execute[[S2]]$

In either case, CASL allows a usage closer to that of conventional denotational semantics, with the double brackets employed both in the left- and right-hand sides of the equations.

Thus it seems that CASL is well-suited for specifying semantic functions by semantic equations.

7 Specifying Semantic Entities

One major advantage of CASL would be for specifying abstract sorts equipped with constructors and selectors, which are typically used to represent semantic entities such as arrays and procedures. So-called *datatype declarations* in CASL (reminiscent of datatype definitions in Standard ML) allow such specifications to be written very succinctly. For instance, an abstract datatype of pairs can be specified as follows:

free type $Pair ::= pair(left : X; right : Y)$

The declarations of the constructor operation $pair : X \times Y \to Pair$, the selector operations $left : Pair \to X$ and $right : Pair \to Y$, and the axioms defining the selectors, are all implicit in the above specification. When there is more than one constructor, the selectors are usually partial, e.g.:

free type $List ::= nil \mid cons(head :?Item; tail :?List)$

although partiality may alternatively be swept under the carpet by restricting the selectors to subsorts:

free types $List ::= nil \mid sort\ NonEmptyList;$
$NonEmptyList ::= cons(head : Item; tail : List)$

When the keyword '**free**' is omitted in a datatype declaration, extensions of the specification may add further constructors and embedded subsorts. Many somewhat tedious specifications of abstract datatypes in action-semantic descriptions could be expressed much more concisely by exploiting the datatype declarations provided by CASL.

In connection with some parts of Action Notation, certain sorts of data have to be specified: *Datum*, *Bindable*, *Storable*, etc. Typically, the elements of such a sort is the union of various other specified sorts, e.g. *Integer*, *Cell*. Using UA as meta-notation, a sort may be simply equated with a union of other sorts, or sort inclusions may be specified. The CASL notation for datatype declarations where the alternatives are embedded subsorts is quite close to UA:

free types $Datum ::= sort\ Integer \mid sort\ Cell$

However, with free datatype declarations in CASL, embedded subsorts cannot have any common elements, in contrast with unions in UA. It would thus be safer to declare subsort embeddings directly in CASL (or at least to remove the freeness constraints when using datatype declarations).

Finally, when reconsidering how the data used in action-semantic descriptions may be specified, the interests of those implementing prototyping and compiler-generation tools for Action Semantics should be taken into account. The unrestricted axiomatic specification style used for specifying datatypes in UA and CASL is appropriate for stating abstract properties of operations, and for reasoning about the consequences of the specification, but it makes it difficult for tools to determine efficient representations of data and to implement the specified operations and predicates. It might be advantageous to restrict to a more explicit definitional style of data specification, with a clear distinction between declarations of constructors and selectors, and with other operations defined (perhaps inductively) using very restricted forms of equations. This would correspond to adopting a "functional programming" sub-language of CASL [6]. The CASL libraries of basic datatypes are currently specified using full CASL, but the relevant ones could probably be re-specified in such a sub-language.

8 Modular Specification

The meta-notation for modules in UA was designed specifically for use in Action Semantics. The main idea is to use conventional section titles as the names of modules, with submodules inheriting all the notation introduced at the enclosing levels (but not that introduced by submodules on the same or lower levels). A module may moreover explicitly import other modules, either for local use or for re-export. and mutual reference between modules is allowed.

In contrast, named modules in CASL are indicated in a more explicit definitional style, with importation of other modules expressed by structured specification terms, for instance:

spec STATEMENTS =
EXPRESSIONS **and** DECLARATIONS **then** ...

Modules in CASL cannot be nested, nor can the common importation of a module by a group of modules be factored out. Even mutual reference between modules is prohibited in CASL: module names have linear visibility (although mutual reference can be simulated by introducing auxiliary modules that merely declare the common symbols, replacing the mutual references by references to the auxiliary modules). The direct use of CASL for expressing the modular structure of action-semantic descriptions might thus be relatively obtrusive and tedious, compared to UA.

9 Tentative Conclusion

Since this paper is of a rather speculative nature, it would be unwise to try and draw any definite conclusions. Nevertheless, it appears from the above considerations that it might be advantageous to use CASL for specifying Data Notation, Action Notation, semantic functions, and semantic entities. On the other hand,

it does not seem desirable to adopt CASL for direct use as a meta-notation for specifying abstract syntax and the modular structure of action-semantic descriptions; CASL (and its associated tools) may however still be used indirectly, by providing a translation to it from whatever meta-notation is used there. Although it has not yet been investigated in detail, such a translation could probably be given fairly easily for a meta-notation close to the Unified Algebra meta-notation presently used for specifying abstract syntax and modular structure in action-semantic descriptions.

References

1. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI`.
2. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [1], Oct. 1998.
3. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (Documents/CASL/Semantics, version 0.96), in [1], July 1999.
4. CoFI Tools Task Group. The Common Framework Initiative for algebraic specification and development: Tools. `http://www.loria.fr/~ hkirchne/CoFI/Tools/`.
5. A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Subsorted partial higher-order logic as an extension of CASL. Note L-10, in [1], Oct. 1998.
6. T. Mossakowski. Two "functional programming" sublanguages of CASL. Note L-9, in [1], Mar. 1998.
7. P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.
8. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
9. P. D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997. Also available at `http://www.brics.dk/RS/97/48/`, `http://www.brics.dk/Projects/CoFI/Documents/Tentative/Mosses97TAPSOFT`.
10. P. D. Mosses. CASL: A guided tour of its design. In *WADT'98*, volume 1589 of *LNCS*, pages 216–240. Springer-Verlag, 1999. Also available at `http://www.brics.dk/RS/98/43/`.
11. P. D. Mosses. A modular SOS for Action Notation. Research Series RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/56`. Full version of [12].
12. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In P. D. Mosses and D. A. Watt, editors, *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at `http://www.brics.dk/RS/99/56/`.
13. M. Roggenbach and T. Mossakowski. Basic datatypes in CASL. Note L-12, version 0.4.1, in [1], May 2000.

Modular SOS and Action Semantics
(Abstract)

Peter D. Mosses

BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark

Abstract. Modular SOS (MSOS) [5] is a variant of Plotkin's familiar Structural Operational Semantics framework. The main idea of MSOS is to use labelled transition systems where the configurations are simply syntax, and the labels on the transitions carry any auxiliary information that is needed, such as environments, store-updates, and signals. New components can be added to the labels without any reformulation of MSOS rules, and a high degree of modularity is obtained.
The Action Notation (AN) used in Action Semantics (AS) was originally defined using SOS. In 1999, the definition of AN was reformulated in MSOS [7]—partly as a major demonstration of the usefulness of MSOS, partly for use in connection with a reconsideration of the design of AN. The MSOS of AN is specified in CASL, the Common Algebraic Specification Language [3, 4, 9].
An MSOS of AN-2, the proposed new version [2] of AN, has been developed, and is currently being polished and checked. The original MSOS rules for many primitive actions and action combinators are being reused in the MSOS of AN-2.
The modularity of MSOS appears to be as good as that of AS regarding independence of the description from details of the processed information. Concerning control flow, however, it is just as tedious to specify control flow in MSOS as in conventional SOS; in particular, axioms for the propagation of exceptions have to be given for each normal construct. In contrast, exceptions in AN are automatically propagated through all action combinators other than those concerned with exception-handling, the modularity of AS for specifying exceptional control flow is significantly better than that of MSOS.

References

1. *AS 2000, Proc. 3rd International Workshop on Action Semantics, Recife, Brazil*, Notes Series, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000. This volume.
2. S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In AS 2000 [1].
3. P. D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997. Also available at http://www.brics.dk/RS/97/48/, http://www.brics.dk/Projects/CoFI/Documents/Tentative/Mosses97TAPSOFT.

4. P. D. Mosses. CASL: A guided tour of its design. In *WADT'98*, volume 1589 of *LNCS*, pages 216–240. Springer-Verlag, 1999. Also available at http://www.brics.dk/RS/98/43/.
5. P. D. Mosses. Foundations of modular SOS. Research Series RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. http://www.brics.dk/RS/99/54; full version of [6].
6. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at http://www.brics.dk/RS/99/54/.
7. P. D. Mosses. A modular SOS for Action Notation. Research Series RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. http://www.brics.dk/RS/99/56. Full version of [8].
8. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at http://www.brics.dk/RS/99/56/.
9. P. D. Mosses. CASL and Action Semantics. In AS 2000 [1].

An Action Semantics for the D2L Design Description Language

Christina von Flach G. Chavez[1,2], Sylvia de Oliveira e Cruz[1], and Sergio E.R. de Carvalho*

[1] Department of Informatics, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil. (flach,sylvia)@inf.puc-rio.br
[2] Department of Computing Science, Federal University of Bahia, Salvador, Brazil. flach@dcc.ufba.br

Abstract. D2L is a language for expressing object-oriented designs. It provides mechanisms for the representation of classes and their relationships, object life-cycles, scenarios, state-dependent behavior and object communication in general, as well as some less conventional features, such as several new semantics for object behavior and some predefined automatic relationships between objects. D2L is currently inserted in a CASE tool, through which methodology-independent designs can take place.
This paper reports the work in progress concerning the development of a complete formal specification for D2L using Action Semantics. The first stage of this work comprises the formal specification of D2L dynamic semantics. Up until now, the effort to formalize D2L has motivated some relevant questions about the language's semantics and also about the use of Action Semantics to formalize some features.
With the complete action-semantic specification for D2L, we hope to provide an executable model for designs expressed in D2L that can be simulated and also automatically translated to source code in some object-oriented languages.

1 Introduction

In the last decade, several object-oriented development methods appeared, each with its own terminology, notation and processes. Since these methods have been built around the same concepts, some attempts to unify them emerged naturally.

D2L (*D*esign *D*escription *L*anguage) [Car97] is a language for expressing object-oriented designs. It reifies not a new methodology or unification of features, but rather a carefully organized object-oriented discourse, which can be visually and textually realized. D2L has been designed to be the textual realization for such object-oriented discourse. It is currently embedded in 2GOOD (*2*nd *G*eneration *O*bject *O*riented *D*evelopment) [CCO98], a CASE tool that supports graphical descriptions of object-oriented designs (discourse's visual realization), providing mechanisms to represent classes and their relationships, object communication, state-dependent behavior and scenario diagrams, as well as

* *in memoriam*

a framework for transformational development from design to implementation. The fact that both graphical design tool and textual design description language reflect exactly the same object-oriented discourse promotes development without discontinuities, documentation, and traceability.

In its original purpose, D2L code was expected to be automatically obtained from 2GOOD designs, and also automatically transformed into some object-oriented language source code, to be eventually compiled, generating executable code. This second transformation step was instantiated with C++ as the target object-oriented language, and was implemented using TXL [CCH95], a semantic-preserving transformational system. This configuration was experimented in pilot projects but it was noticed a kind of programmer's resilience with respect to writing code using an intermediate language. Moreover, TXL execution was very inefficient (both time and space-consuming): the transformation of a D2L program with 10 classes (a small number for a real-life project) was almost impracticable. Finally, the generated C++ code was quite unreadable. The need for a D2L compiler was rather obvious. Nevertheless, D2L does not have an implemented compiler yet.

In the last few months, new extensions have been proposed to 2GOOD/D2L, mainly to provide design-level support to aspects [CL00a] and extensions to pronoun-based communication [CL00b]. These two parallel efforts are centered around D2L and require a complete and unambiguous formal description of the language. Besides, we are interested in providing an executable model for designs expressed in D2L that can be not only translated to source code in some object-oriented languages (including C++), but also simulated prior to translation. Again, a formal description of D2L can be very useful. Having these goals in mind, the formalization of D2L turned out to be an intrinsic part of both efforts [CL00a,CL00b].

Although Denotational Semantics has been the most popular tool among programming language researchers for studying and describing several aspects of programming languages, we have adopted the Action Semantics framework to develop the formalization of D2L. Actions Semantics provides a high-level notation to formally describe the semantics of programming languages, which gives a novice the intuition about what is being formalized. Furthermore, Action Semantics specifications are compositional and modular. Hence, they can be straightforwardly modified to reflect language design changes, and to reuse parts of an existing language definition in the specification of similar, related languages. With these features, the specification of the new proposed extensions may be easily incorporated into D2L specification, with low impact over the previous specification.

This paper is structured as follows. In section 2 we provide a brief description of D2L. Section 3 presents some parts of the current specification stage. In section 4, we present some specification problems found up until now and in section 5, we compare our work to previous similar approaches. Finally, in section 6 we discuss the actual specification stage and the next steps to complete the specification.

2 Overview of D2L

D2L is a textual design description language for object-oriented systems, whose main purpose is to minimize discontinuities from design to code. D2L is a strongly-typed, concurrent, pure object-oriented language. It provides common features found in ordinary object-oriented languages, such as classes, inheritance and polymorphism. Less conventional features are also provided by D2L. For example, D2L objects are polymorphic and can satisfy requests in several different ways: procedurally, answering asynchronous, handshake and future messages, cooperatively and exceptionally. Concurrency in an inherent aspect in D2L. Other D2L features include automatically created relationships among objects, denoted by pronouns, and the textual realization of state-dependent behavior.

D2L design criteria include simplicity and familiarity. The language is small and most of its object-oriented constructs can be explained in terms of well-known constructs provided by other object-oriented languages.

In the present section, we elaborate only those features that will be covered in section 3.

2.1 Programs

From a static point of view, a D2L program is a collection of possibly nested classes, declared in the program or brought in from libraries. There is no external referencing environment (no main program, external data or functions). This is a pure object-oriented environment.

Class collections can be constructed for execution or for storage in libraries. D2L applications are essentially brokered [BMR+96]: an executable program must have at least one top-level class containing a procedure called "Main". One of these classes, indicated via the execution environment, is the *main class* in the program. On the other hand, a class library does not need (but may contain) one or more classes with defined "Main" procedures.

From a dynamic point of view, a D2L program is a collection of concurrent interacting objects. The execution of a program begins with the application of the "Main" procedure to an automatically created object of the *main class*.

2.2 Classes and Interfaces

Classes model objects, prescribing their structure and behavior. Interfaces do not model objects; they only define behaviors to be implemented by classes.

Classes can be concrete, abstract or generic. Abstract classes allow the declaration but not the creation of objects, since at least one of its operations is not fully defined. These classes are useful in inheritance hierarchies, promoting polymorphism. Concrete classes allow the declaration and creation of objects; they may inherit from abstract classes (but not vice-versa), implementing incomplete operations, and thus allowing the creation of objects. Generic or parameterized classes are class models, which can be instantiated with argument classes.

```
CLASS SuperFoo<S>;
  ...
END CLASS
```

```
INTERFACE IntFoo1;
  ...
  PROCEDURE p1;
  ...
END INTERFACE
```

```
INTERFACE IntFoo2;
  ...
  PROCEDURE p2;
  ...
END INTERFACE
```

```
CLASS AbstractFoo;
  ...
  PROCEDURE p;
END CLASS
```

```
CLASS Foo;
  OBJECT
    SuperFoo<INT> f;
  END OBJECT
  ...
  PROCEDURE p;
  ...
  END PROCEDURE
  ...
END CLASS
```

```
CLASS GenericFoo <S,T>
INHERITS SuperFoo<S>
IMPLEMENTS IntFoo1,IntFoo2;
  ...
  PROCEDURE p1;
  ...
  END PROCEDURE
  PROCEDURE p2;
  ...
  END PROCEDURE
END CLASS
```

Fig. 1. An Example

D2L provides simple inheritance between classes, creating a super-type/subtype relationship, and multiple inheritance between classes and interfaces, creating interface/implements relationships.

Figure 1 presents several examples of D2L classes. `Foo` is a concrete class with a member `f` of type `SuperFoo<INT>`. The class `SuperFoo<INT>` corresponds to the generic class `SuperFoo<S>`, with every ocurrence of `S` replaced by `INT`. The generic class `GenericFoo` inherits from class `SuperFoo`, introducing one more generic parameter `T`; parameter correspondence is nominal, not positional. The class `GenericFoo` also implements two interfaces: `IntFoo1` and `IntFoo2`. `AbstractFoo` is an abstract class, since it declares a procedure named `P` with no defined body.

A class declaration may optionally contain the following members:

- nesting: a class A may be contained in a class B, having full visibility of its features, and being protected from accesses external to B;
- object structure: a record of attribute declarations, each containing the name of an attribute and its modeling class name. Corresponding to each object there exists, at execution-time, a block of data for its attributes. At object creation time, a default initial value is assigned to each object. Object components are automatically protected from external accesses, being visible only to class operations. This protection can be explicitly relaxed by programmers;
- common structure: another record, statically allocated, unique and visible to all class objects, through class operations; direct access to this structure is denied to class users, but may be explicitly given by programmer's option;

– state component: a special object component, modeling the state-dependent behavior of class objects;
– class constants: read-only class objects;
– class messages: declarations of messages that class objects may send to pronouns;
– class exceptions: declarations of exceptions that class objects may propagate;
– object behavior: declarations of the operations applicable to class objects.

2.3 Objects

Objects may be declared as members of other objects, in the class common area, locally in operations, as operation parameters, and as operation results. They can be collected in containers, such as arrays and lists (objects of instantiated generic classes).

All D2L objects are polymorphic and may change class at execution-time (always in the hierarchy rooted by their declaration classes). These objects are heap allocated, and their life cycles are programmer controlled.

Each object is controlled by a *monitor*, in charge of handling several aspects relating to its behavior, as for example message receiving, guard evaluation, and operation scheduling [CCO00].

The modeling of an object's monitor starts when the monitored object is created. At this time, the monitor acquires as components a descriptor and a message queue to be used at execution time. The descriptor will record information such as the objects current class. The message queue will hold the operation requests the object receives. The monitor intercepts all object messages, organizing them in its message queue, and sends to the object the next message to be executed. If no operation is found that corresponds to a message, perhaps due to some polymorphic transformation suffered by the object, then the message is lost. The monitor needs to control a thread and an execution stack to fulfill the expected operation concurrency.

Real time features, guard evaluation, message priorities and exception handling are all part of D2L. They are not described here due to space considerations.

Pronouns. Certain automatic relationships between objects are created at execution-time: *creator-creature* and *parent-son*. The creator is an object that, while servicing a request, creates another object, its creature. The parent is an object that, in its structure declaration, contains the declaration of its son. These relationships are represented in D2L by the generic designators CREATOR and PARENT, respectively. Using these designators, class decoupling can be increased, if classes model objects that relate to the outside world by sending asynchronous messages to their creators or parents (their clients).

The automatic creation of parent-son and creator-creature relationships relieves programmers of complicated code that must be written to allow generic references of this kind (as recommended in the Composite and Chain of Responsibility patterns [G$^+$95]). It also alleviates the restriction that objects must know the identifiers of their message-receiving collaborators.

2.4 Object behavior

D2L objects can satisfy requests in several different ways: procedurally, answering asynchronous, handshake and future messages, cooperatively, and exceptionally. Special syntax and semantics exist for expressing this varied object behavior; it is up to designer to select the most appropriate in each situation.

The procedural or methodical behavior is that found in other object-oriented languages: the clients execution is suspended until the invoked procedure execution terminates.

Objects in D2L may receive asynchronous messages, messages with handshake protocols, or messages with future synchronism. Asynchronous message receiving does not block the client; handshake message receiving blocks the client until the reception is acknowledged; future message clients may self-block to receive parameters from the server object.

Coroutines and iterators can also be declared in classes, offering a cooperative behavior for objects. Iterators are associated to loops, controlled by objects whose next values are computed outside the loop, in the iterator provided by their modeling classes. Threads are also useful here, mainly to save local execution environments and resume addresses.

Finally, to handle exceptional conditions that may arise during program execution, D2L allows for the definition of exception handlers at the operation level and in classes. Class exception handlers specify another handling scope, being automatically applied to the receiving object if no local handler is defined for some raised exception.

2.5 State-dependent behavior

In object-oriented systems, the behavior of an object, when responding to a stimulus, may depend on the values of its components at that time, or on its state. Several object-oriented methods use states and transitions to define object behavior, when this behavior depends on the past history of stimuli received by objects (see Schaler-Mellor [SM94], ROOM [SGW94], Rumbaugh [R$^+$91], Booch [Boo95]).

A state is actually an abstraction of the past requests made to an object, and transitions indicate behavior requests (they usually cause state changes). A state transition diagram [Har87] or table is used in most methods to represent valid sequences of such stimuli. To describe this feature textually, D2L adapts the State pattern described in [G$^+$95].

The State pattern consists basically in representing diagrams' states as classes in an inheritance hierarchy, and diagrams' transitions as method requests. A *state component* is an instance of the root class in this hierarchy. Any request sent to an object that contains a state component is delegated to it. This state component plays the role of navigating in this hierarchy, i.e., it is responsible for the implementation of its parent's state-dependent behavior.

3 D2L Action Semantics

In this section, we present the first results related to the provision of a specification for D2L dynamic semantics using Action Semantics.

3.1 Classes

As stated in section 2.2, D2L classes can be concrete, abstract or generic. In Java, a class declaration introduces a new reference type, which denotes a class. Java Action Semantics [BW99] specifies that class declarations are elaborated through the semantic function `elaborate` that binds each declared class name to a value of sort `class`. In D2L specification, generic classes motivated us to adopt a distinct approach to elaborate class declarations. A generic class is a class template that when used with actual parameters, as for example, in object declarations, instantiates a new concrete class with its class parameters bound to the given actual parameters. The reference to the generic class must be kept as well as the reference to the instantiated class. The solution we propose in D2L Action Semantics is to encapsulate each class declaration as a *class abstraction*.

```
class-abstraction = abstraction [giving a class]
                                [using the given type*]
```

A class abstraction is an abstraction that, when enacted for the first time, introduces a new reference type, denoted by a class. Further enactions of a previously enacted class abstraction give the corresponding introduced reference type. The `given type*` yields the actual parameters that will be bound to class parameters, for generic classes.

The following sort describes a D2L class in Action Semantics:

```
class = class of (derived-name, interface-bindings,
                  nested-classes, class-variable-bindings,
                  instance-variable-allocator,
                  state-component-allocator, class-constants,
                  class-messages, class-exceptions,
                  method-bindings, class?)
```

- `derived-name` specifies the name of the class (suffixed by the names of the classes used as actual parameters, if the class is generic).
- `interface-bindings` is a mapping for the direct superinterfaces of the class;
- `nested-classes` is a mapping for the nested classes of the class;
- `class-variable-bindings` is a mapping for the class variables of the class;
- `class-constants` is a mapping for class constants;
- `class-messages` is a mapping for class message signatures of the class;
- `class-exceptions` is a list of class exceptions names;
- `method-bindings` is a mapping for the method abstractions of the class;
- `instance-variable-allocator` and `state-component-allocator` are abstractions that, when enacted, allocate storage for the instance variables and for the state component of the class, respectivelly;
- `class` is the direct superclass of the defined class.

Class Elaboration. Class elaboration is performed by a semantic function that takes class declarations as arguments and gives an action that binds each declared class name to a class abstraction.

```
elaborate _ :: Class-Declaration ->
                   action [binding]
                          [using current bindings]

(*) elaborate [[ "CLASS" S:Class-Specifier
                 H:Inheritance-Clause
                 P:Implements-Clause?
                 B:Class-Body? "END" "CLASS" ]] =
       bind the simple name of S to the class-abstraction of (
         the closure of the abstraction of (
         furthermore
         | actualize class parameters of S              (1)
         thence
         | | give the derived-name of S                 (2)
         | then
         | | | give the class bound to                 (3)
         | | |   the given derived-name
         | | or
         | | | | check not (there is a class bound     (4)
         | | | |   to the given derived-name)
         | | | and then
         | | | | | | give the given derived-name and
         | | | | | | enact (application
         | | | | | |  (the class-abstraction bound to the
         | | | | | |   simple name of H) to the arguments of H)
         | | | | | then
         | | | | | | recursively bind the given derived-name#1
         | | | | | | to the class of (the given derived-name#1,
         | | | | | |                  ...  , the given class#2)
         | | | | and then
         | | | | | give the class bound to the given derived-name)).
```

Fig. 2. Class elaboration

Figure 2 presents the semantic function `elaborate`[1] defined for class declarations. The sequence of actions encapsulated by a class abstraction implements the following steps:

1. The class parameters defined in `S` are bound to the classes given as actual parameters;

[1] In the semantic equation (*), we suppose the inheritance clause is always present. Assume there is also an equation where the inheritance clause is not present.

2. The derived name of the class is established. For example, the generic class `GenericFoo<S,T>` has a simple name, `GenericFoo`. The instantiated class `GenericFoo<INT,REAL>` has a derived name, `(GenericFoo, (INT, REAL))`.
3. If there is a class bound to the class derived-name, this class is returned.
4. If there is not a class bound to the class derived-name, a new class is created and then returned.
   (a) The enaction of the class abstraction bound to the class' superclass (declared in the inheritance clause `H`) applied to the superclass' arguments may introduce its recursive instantiation, if it has not been instantiated yet, or returns the class bound to the superclass derived-name. `the arguments of H` yields a list of the actual parameters bound to the class parameters names declared in the inheritance clause `H`.
   (b) The class members declared in the class body are elaborated; details are omitted here, but the specification of class members elaboration is quite similar to the one found in [BW99].
   (c) The derived-name of the class is bound to its corresponding sort `class`.

The results of the elaboration of the classes presented in Figure 1 are given below.

Source Action

```
elaborate [[ CLASS SuperFoo<S> ; ]]    before
elaborate [[ CLASS GenericFoo<S,T>
             INHERITS SuperFoo<S> ; ]] before
elaborate [[ CLASS Foo ; ]]
```

Bindings

```
{ SuperFoo -> abstraction ... }
{ GenericFoo -> abstraction ... }
{ Foo -> abstraction ... }
```

Enaction of a class abstraction occurs every time a class name is used, as for example, in object declaration (see next section); it gives a new named reference type or the type previously bound to the corresponding class name.

3.2 Objects

Objects in D2L are inherently concurrent dynamic entities. In D2L Action Semantics, each object is modeled as a sort `object`.

```
object = object of (class, agent, creator)
```

In the sort `object`, we have the following elements:

– `class` is the modeling class of the object;

– `agent` corresponds to an agent subordinated to the object. Storage for the object's instance variables is allocated in the storage associated to this agent. The object's message queue corresponds to the agent's message buffer;
– `creator` is another object that, servicing some request, has created the object.

The reference to `creator` in the sort `object` is redundant, since it must be defined in the agent's local bindings. It is used there for documentation purposes.

Object Declaration. Object declarations allocate storage that will hold references to objects, but do not allocate storage for objects themselves; they must be explicitly created. The syntax for object declaration consists of a class name (denoting the modeling class) followed by an identifier (denoting the variable that will hold the object reference).

Object declaration is performed by a semantic function `elaborate` that takes an object declaration as its argument and gives an action that binds the given identifier to a variable. This variable is allocated to hold a reference to an object or a value of some predefined class (INT, REAL, etc.).

```
elaborate _ :: Object-Declaration ->
       action [binding | storing]
              [using current bindings | current storage] .

(*) elaborate [[ C: Modeling-Class-Name I: Identifier ]] =
   | | elaborate the type denoted by C
   | then
   | | allocate a variable of the given type
   | |   initialized to the default-value of the given type
   then
   | bind the token of I to the given variable .
```

The action `elaborate the type denoted by C` (defined in the semantic function `elaborate` given above) corresponds to the enaction of the class abstraction bound to `C`, applied to the given arguments, if any, and gives a `class`.

```
elaborate the type denoted by _ :: Modeling-Class-Name ->
    action [binding | giving type] [using current bindings]

(*) elaborate the type denoted by
      [[ I: Identifier "<" C:Class-Instances ">" ]] =
        enact (application (the class-abstraction bound
                            to the token of I)
               to the actual parameters of C) .
```

The next example shows the outputs after performing the action `elaborate the type denoted by` applied to `GenericFoo<INT, REAL>`.

Source Action

```
elaborate the type denoted by [[ GenericFoo<INT, REAL> ]]
```

Bindings

```
-- bindings for SuperFoo, GenericFoo, ...
(SuperFoo,INT) -> class of (...)
(GenericFoo,(INT,REAL)) -> class of ((GenericFoo,(INT,REAL)),
                                     ... , (SuperFoo, (INT)) )
```

Transients

```
class of (...) -- the class bound to GenericFoo<INT, REAL>
```

Object Creation. Objects in D2L must be explicitly created, using the operation `CREATE`. During object creation, an agent is subordinated to it. The object's agent receives an offer for a contract containing a protocol that defines all the distinct object behaviors provided in D2L (asynchronous, future and handshakes messages, coroutines, etc.).

Object creation is performed by a semantic function `evaluate` that takes an expression for object creation as argument (a class name followed by the keyword `CREATE`) and gives an action that creates an instance of the given class name, denoted by an `object`.

```
evaluate _ :: Object-Creation ->
    action [giving an object | ... | communicating]
           [using current buffer]

(*) evaluate [[ C:Modeling-Class-Name "CREATE" ]] =
    | elaborate the type denoted by C
    then
    | | | give the given type
    | | and
    | | | | | | offer a contract [to any agent]
    | | | | | |   [containing abstraction of object-behavior]
    | | | | | then
    | | | | | | receive a message[containing an agent]
    | | | | and
    | | | | | give the instance-variable-allocator of given type
    | | | then
    | | | | | send a message [to the agent yielded by
    | | | | |   the contents of the given message#1][containing
    | | | | |   the given instance-variable-allocator#2]
    | | | | and
    | | | | | give the agent yielded by
    | | | | |  the contents of the given message#1
    | then
    | | give the object of (the given type#1, the given agent#2,
    | |                     the object bound to the self-token) .
```

Figure 3 presents `object-behavior`, the protocol offered to objects' agents. While performing this protocol, each subordinated agent:

1. Sends its identification to the contracting agent;
2. Waits for an abstraction corresponding to the instance-variable-allocator;
3. Enacts this abstraction, allocating storage for the object's instance variables;
4. Produces agent's local bindings, and finally,
5. Executes a loop that continuously waits for a message containing a request for a service with optional arguments, and then selects the suitable protocol depending on the kind of the request received.

```
method = procedure | async | handshake | future |
         exception | coroutine | iterator .

object-behavior = action .

(*) object-behavior =
    | | send a message [to the contracting-agent]              (1)
    | |                [containing the performing agent]
    | | and then
    | | receive a message [from the contracting-agent]         (2)
    | |                   [containing an abstraction]
    | then
    | | enact the abstraction yielded by                       (3)
    | |   the contents of the given message
    before
    | | | bind self-token to the performing agent              (4)
    | | before
    | | | bind creator-token to the contracting agent
    | hence
    | unfolding                                                (5)
    | | | receive a message [from any agent]
    | | |   [containing msg(method, arguments)]
    | | then
    | | | | check (method in msg is handshake) then
    | | | | send a message [to the contracting-agent]
    | | | |                [containing the ack-signal]
    | | | | and then  ...
    | | | |  ... << the behavior of handshake method >>
    | | | | then unfold
    | | | or ... << other behaviors  >>
```

Fig. 3. The object-behavior protocol

An Example: Handshake. Suppose that, during its execution, `obj1` sends a handshake message to `obj2`: `obj2 <- handshakeMethod`. The handshake protocol states that the sender `obj1` is supposed to wait for an `ack-signal` from the receiver `obj2` in order to proceed.

The Action Semantics description for the handshake protocol on the client's side (in this case, `obj1`) is given in Figure 4. The action `receive a message [from the agent of the given object#1] [containing the ack-signal]` is specific to the handshake protocol on the client's side; in fact, it must be embedded in a multiple `or` action combinator (similar to the one defined in `object-behavior`, that checks for each defined behavior in D2L, and chooses one, based on the type of the message that has been sent).

```
evaluate _ :: Application -> action
                 [giving a value? | ... ]
                 [using current bindings ... ]

(*) evaluate [[ T:Application-Target "<-"
                O:Operation-Name ";" ]] =
      | | give the receiver denoted by T
      | then
      | | | give the given object
      | | and
      | | | | give the class of the given object
      | | | then
      | | | | give the method bound to the token of O
      | | | | in the given class
      then
      | | send a message[to the agent of the given object#1]
      | |               [containing msg(the given method#2)]
      | and then
      | | receive a message [from the agent of the given object#1]
      | |                   [containing the ack-signal]
      ...
```

Fig. 4. Method Call

4 Specification Problems

The communicative/hybrid facets (agents and send/receive actions) have been quite suitable to model the behavior of D2L objects. Agents provide the desirable intrinsic asynchronism demanded by D2L and the agent's message buffer plays the role of the object's message queue. The semantics of the `receive` action (busy-waiting semantics), however, has imposed some difficulties in modeling the various kinds of synchronous communication provided by D2L.

The modeling of the well-known procedural synchronous behavior, for example, has presented some problems. In a straighfoward implementation for this behavior, the sender object remains blocked (after the synchronous message dispatch) until it receives a "end" message from the receiver object. Within this implementation, however, in case the receiver object sends, directly or not, other

synchronous message to sender object, this message will not be received, because the sender object is blocked waiting for the original "end" message. This behavior is not the desired one, of course, since ordinary synchronous communication (by message exchange between objects) shall simulate procedural or methodical behavior, where there is no blocking, but simply pushing in a execution stack. On the other hand, freeing the synchronous message sender object to receive any other messages, corresponds by no means to the desired behavior, since we do not want to allow this object to receive, from a third object, any message that is not concerned to the current execution line.

The provision of a `receive` action with no blocking at all (as promised by the Action Notation 2) may facilitate our work, but does not solve the specification problems associated to all D2L synchronous behaviors.

The second problem we have approached refers to class binding. The solution provided to the generic class instance problem, using class abstractions and class late-bindings, has given rise to another kind of problem: the binding of a class name to a recently created `class` entity is done in the local bindings of the agent of the first class instance's creator. This binding and the class variables allocation, however, should be done in a global scope and not in the current local (of agent) scope. This global scope will have to be specified as a separate agent; the class global bindings are all expected to be there. Class variables, in turn, will have to be stored in separate agents, one per class, modeling shared memory among class objects. This agent proliferation further complicates system synchronization and storage management.

5 Related Work

In [Mos96b], Mosses provides an action-semantic description for a subset of the Ada language, where tasks and *rendez-vous* are modeled using the communicative facet. Mosses also reports the use of Action Semantics to describe concurrent languages in [Mos96a].

As stated in section 3, the `class` definition adopted in D2L specification is very similar to the one found in the JAS project [BW99]. The main difference is the class binding time. Since Java does not have generic classes, classes can be bound and their class variables allocated at declaration time. Consequently, classes and their variables are statically created; they already exist when the main object is created. In D2L, classes are late-bound at object creation time. A class and its class variables are created when its first object is created.

The JAS current stage does not approach concurrency aspects, disregarding executions in parallel generated for `java.lang.Thread` objects. This simplification is not useful here since concurrency is an inherent aspect in D2L.

In [Rep91] the standard ML language is extended with the introduction of mechanisms for spawning new processes and for synchronous exchange of values between these processes over typed channels. In [MM94], this extension is described with Action Semantics and analogous to D2L, agents are used in parallel processing. The ML proposal, however, is to provide a low-level language with

primitives that allow programmers to control threads and channels. In D2L these features are built-in, so it is up to the language to manage and to have control over them. Furthermore, D2L descriptions may be richer and more complicated due to the several distinct semantics provided for object behavior.

In [PMM95] a parallel object-oriented language – POOL – is described using Action Semantics. Like D2L, this language has integrated parallelism in the object-oriented model by supplying each object with a local independent process that is executed in parallel with all the other objects in the system. The communication mechanism in POOL, however, is only synchronous, unlike D2L, where asynchronous communication is provided. POOL objects are agents and the communication between these agents is made via message exchange as in D2L. However, the *busy-waiting* problem presented in section 4 is not mentioned in [PMM95].

6 Conclusions

This paper introduces D2L, a textual design description language for object-oriented systems and presents the first results related to the specification of the dynamic semantics of D2L classes, objects and object behavior using Action Semantics. The main goal of this work is the provision of a complete formal specification for D2L.

Up until now, the contributions of our effort to formalize D2L using Action Semantics are the specification of class declarations using class abstractions, the specification of concurrent objects using agents and also the specification of the several object behaviors provided by D2L. Furthermore, we anticipate that the formalization of a language for expressing object-oriented designs can be very useful to software development as a whole.

The specification of D2L using AS has given us some insights about many language aspects, as well as design decisions that possibly had not been explicitly stated before.

Surprisingly, the task of formalizing D2L using Action Semantics, guided through some useful examples [Mos96b,BW99], has been amazingly easy, especially considering the authors do not have a solid background or previous relevant experience on formalization.

6.1 Future Work

As part of the current specification stage, we expect to be able to describe the whole set of semantics for object communication provided by D2L. The proposed changes in the forthcoming version of Action Notation (AN-2) [LMW00] seem to facilitate our job and to provide solutions to the problems stated in section 4.

In AN-2, message reception is fair and messages are tagged. This sort of non-determinism seems more appropriate to languages with intrinsic asynchronism such as D2L. The primitive actions for explicit agent creation and agent destruction resembles the computation model. Besides, transients passing seems to be

less expensive in AN-2, generating more concise specifications, a very important characteristic for languages with rich semantics such as D2L.

To complete the description of D2L dynamic semantics, the following features must be specified: nested classes, visibility, state-dependent behavior, state component inheritance, class constants and exceptions. These features have not been approached in the current stage of D2L specification, but we hope that their introduction will not cause undue difficulties, since other languages' similar features have been modeled using Action Semantics (such as inner classes and access modifiers in Java, for example).

The next stage of this work will comprise the specification of D2L static semantics. After a reliable version of D2L Action Semantics is developed, i.e, verified using ASD or related tools [BMW92,RAT], we will be able to extend D2L with the features proposed in [CL00b] and [CL00a], and give their formal specification using Action Semantics.

Acknowledgements

We wish to thank Prof. Lucena, our advisor, for having "adopted" us without any restrictions, as well as for keeping the work initiated with Prof. Sergio Carvalho in our post-graduate (PhD) programme.

Finally, we dedicate this work to the D2L designer, Prof. Sergio Carvalho, whose bright ideas and remarkable contributions in the field of object-oriented systems, we have the honor (and audacity) to try to keep alive.

References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stahl. *Pattern-Oriented Software Architecture*. Wiley, 1996.

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.

[Boo95] G. Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, 2nd edition, 1995.

[BW99] Deryck F. Brown and David A. Watt. JAS: a Java Action Semantics. In *Proc. of 2nd International Workshop on Action Semantics*, pages 43–55, Amsterdam, March 1999. BRICS Notes Series.

[Car97] Sergio E. R. Carvalho. DDL: An Object-Oriented Design Description Language. Technical Report PUC-RioInf.MCC29/97, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Department of Informatics, Rio de Janeiro, Brazil, 1997.

[CCH95] James R. Cordy, Ian H. Carmichael, and Russel Halliday. *The TXL Programming Language*. Legasys Corp., 1995.

[CCO98] Sergio E. R. Carvalho, S. O. Cruz, and T. C. Oliveira. Second Generation Object-Oriented Development. Electronic Notes in Theoretical Computer Science. WWW[2], 1998.

[2] http://www.elsevier/nl/locate/entcs/volume14.html

[CCO00] Sergio E. R. Carvalho, Sylvia O. Cruz, and Toacy C. Oliveira. Concurrency Features in 2GOOD/DDL. In *Proc. IDEAS 2000*, pages 100–111, Cancun, Mexico, April 2000.

[CL00a] Christina von Flach G. Chavez and Carlos J. P. Lucena. Design-Level Support for Aspects. Ph.D. Thesis Proposal, Pontifical Catholic University of Rio de Janeiro, Department of Informatics, 2000.

[CL00b] Sylvia O. Cruz and Carlos J. P. Lucena. Pronoun-based Object Communication. Ph.D. Thesis Proposal, Pontifical Catholic University of Rio de Janeiro, Department of Informatics, 2000.

[G+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming 8*, pages 231–274, July 1987.

[LMW00] Soren B. Lassen, Peter D. Mosses, and David A. Watt. AN-2: Revised Action Notation – Informal Summary. DRAFT PROPOSAL AN-2/v0.6, 2000.

[MM94] Peter D. Mosses and Martín A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *Lecture Notes in Computer Science*, pages 461–479. Springer-Verlag, 1994. WWW[3].

[Mos96a] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996. WWW[4], FTP[5].

[Mos96b] Peter D. Mosses. A tutorial on action semantics. 50pp. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996). WWW[6], FTP[7], March 1996.

[PMM95] Giovanny L. Palma, Martin Musicante, and Silvio R. L. Meira. A Novel Formal Semantics for a Parallel Object-Oriented Language. In *Proc. of XV Intl. Conf. Of the Chilean Computer Science Society*, Arica, November 1995.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RAT] Recife Action Tools. WWW[8].

[Rep91] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley, 1994.

[SM94] S. Schlaer and S. Mellor. *Object Life Cycles: Modeling the World in State*. Yourdon Wiley, 1994.

[3] http://www.brics.dk/RS/94/20/
[4] http://www.brics.dk/RS/96/53/
[5] ftp://ftp.brics.dk/Projects/AS/Papers/Mosses96MFCS
[6] http://www.brics.dk/NS/96/14/
[7] ftp://ftp.brics.dk/Projects/AS/Papers/Mosses96DRAFT
[8] http://www.di.ufpe.br/∼rat

An Action Semantics for STG

Francisco Heron de Carvalho Junior, Hermano Perreli de Moura, Ricardo Massa Ferreira Lima, and Rafael Dueire Lins

Federal University of Pernambuco
Centre of Informatics
Recife, Brasil
{fhcj,hermano,rmfl,rdl}@cin.ufpe.br

Abstract. STG (*Shared Term Graph*) is a very simple higher order non-strict pure functional language, primarily designed to be the "abstract machine code" of STG-machine (*Spineless Tagless G-Machine*), an abstract machine designed to support implementation of *lazy* functional languages, like *Haskell*, on stock hardware[13]. GHC[14] and *Haskell*/$\mu\Gamma$CMC[11] are compilers for *Haskell* that use STG as intermediate code. This paper gives an action semantics for STG, trying to capture its essential characteristics, like *lazy evaluation*, *higher order functions*, *pattern matching by case expressions* and *support for algebraic values*.

1 Introduction

Action semantics is a very useful formalism to describe the meaning of programmming language concepts [7], with some desirable properties like *clarity*, *modularity*, and *extensibility*[5]. Action notation allows us to specify fundamental aspects of programming languages in a closer way to their implementation mechanisms, but without losing power of expressing the meaning of programs in a high level of abstraction. This is due to its intermediary level of abstraction, between *denotational* and *operational semantics*. Higher order non-strict pure functional languages have a straitforward denotational semantics. However, using this formalism, some of their control features should be described in some specific style, like *continuation passing style*, which makes language specification hard to modify and extent in the design process [5]. One of the primary goals of a development of action semantics formalism has been to remedy this problem.

STG is a realistic functional language used as intermediate code by some functional language compilers. In spite of its simplicity, STG supports many important features that appear in an actual higher order non-strict pure functional languages, like *bindings*, *lazy evaluation*, *higher order functions*, *algebraic values*, and *pattern matching*. Thus, a STG action semantics specification can be used as a starting point for defining an action semantics specification for a higher level, more complex, functional language that supports these features, like *Haskell*, due to the high degree of *modularity* and *extensibility* of the action semantics formalism. The possibility for automatic generation of compiler prototypes from a semantic specification is another desirable feature of action

semantics formalism, allowing evaluation of semantic and syntax features of new designed languages. With that, there is some tools available. See [1] for a survey.

In addition to this introduction, in this paper there are three more sections. In Sect. 2, we talk about STG language, focusing on its main characteristics and on describing its use as intermediate code by some functional language compilers. In Sect. 3, we present the action semantics specification defined in this work for STG, showing how its main characteristics were specified using action notation. Finally, in Sect. 4, we present some conclusions of this work and give some lines for future work. The bibliography is presented in the end of the document.

2 The STG Language

The STG (*Shared Term Graph)* language was originally designed to be the "abstract machine code" of the STG-machine (*Spineless Tagless G-machine*), a virtual machine that supports implementation of *lazy* functional languages on stock hardware. STG is a very austere pure non-strict functional language with a formal *operational semantics* expressed as a state transition system, as well as the usual *denotational semantics*. This is the main characteristic that distinguish STG from other languages used as intermediate code by functional language compilers[13].

STG has many salient characteristics that are supported only for improving efficiency of programs when running under STG-machine. For example, its syntax exposes when a closure is constructed and whether it is updated, usually implicit matters. In spite of this fact, STG can be seen as entirely independent from STG-machine. However, in account of the importance of these salient characteristics to the action semantics specification presented in this paper, following we talk about some of them, as in [13]:

- *All function and constructor arguments are simple variables or constants.* This unnested syntax (there is not sub-expressions) is a consequence of the operational reality that, in STG-machine, all functional arguments are prepared prior to the call, either by constructing a closure or evaluating them. When translating an usual functional language, like *Haskell*, to STG, non-trivial arguments are represented by *let bindings*. As a consequence, STG programs can be very unreadable from a user point of view. But, in fact, as we implicitly said early, STG is not a language for final programmers, but an intermediate core language in the compiling process of higher level functional languages;
- *All constructors and built-in operators are saturated.* This characteristic is not usual in common higher order languages, but simplifies the operational semantics of STG programs.
- *Pattern matching is performed only by case expressions and patterns are one-level.* This characteristic also simplifies the operational semantics of STG programs. More complex forms of pattern matching can be translated into this form [9]. The *selector expression* of a *case* can be arbitrary, not only a simple variable or constant;

– *There is a special form of binding.* Its general form is:

$$f = \{v_1, \cdots v_n\} \setminus \pi \ \{a_1, \cdots a_n\} \rightarrow \textit{expression}$$

The free variables $v_1, \cdots v_n$ and the update flag π do not have denotational meaning, only operational. Thus, we decided to ignore these syntactic components from the semantic specification. This decision is a mere consequence of our goal on describing only the meaning of STG programs, not how they execute on STG-machine;

– *Support for unboxed-values.* In functional languages, unboxed values [16] are a way for improving performance of numeric calculations, avoiding the indirection generated by storing all primitive values in closures. In STG, all primitive values are *unboxed*.

2.1 GHC Compiler

GHC (*Glasgow Haskell Compiler*) [14] is one of the most popular, complete and efficient free compilers for *Haskell* [15], a pure non strict higher order functional language developed by an international group of researchers since 1987. Now, it became a *de facto* standard for research in functional languages programming and implementation, and for development of real applications under this programming paradigm.

GHC uses STG as intermediate code. In fact, most of program transformations used to improve the performance of functional code, an important characteristic of GHC compiler, is performed over the generated STG code, not over the original *Haskell* code. This simplifies the construction of the compiler and reduces its complexity without loss of optimization opportunities. Also, as we explained early, STG exposes some important usually implicit matters, like when a closure is constructed and when it is updated, useful information for improving performance of functional code.

In Fig. 1, we show the main compilation phases of GHC, as described in [14]. The *Haskell* code is translated to *Core* language code, a very much simpler (*desugared*) version of the original *Haskell* code, over which some optional transformations can be applied for improving it. A simple pass converts the *Core* code to STG code and, again, a variety of optional transformations are applied. The STG code is then translated to *Abstract C* (maximizing *portability*) or, if preferred, to assembly language for a particular machine (maximizing *efficiency*). Finally, if the first one is assumed, a *target code printer* prints abstract C in a form acceptable to a C compiler.

2.2 Haskell/$\mu\Gamma$CMC Compiler

$\mu\Gamma$CMC is an abstract machine for efficient implementation of functional languages, based on Categorical Multi-Combinators[17]. In order to obtain portability and efficiency, it transfers the control of execution flow to C when it is

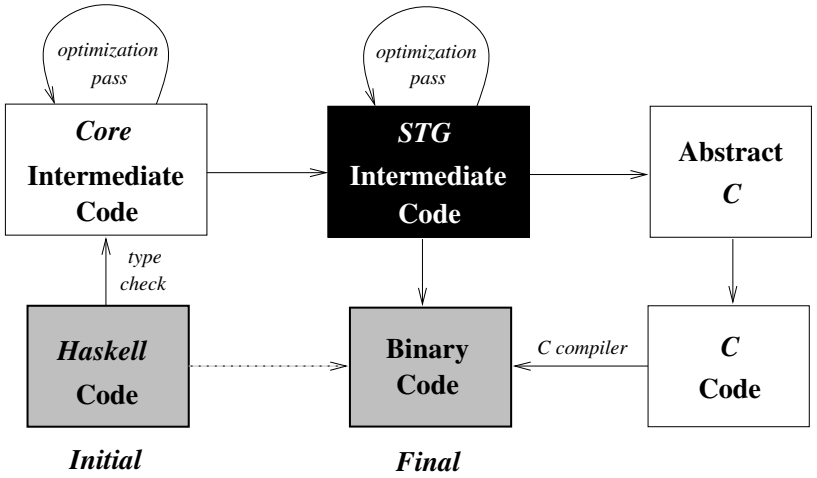


Fig. 1. GHC Compiling Process

possible. The evaliation mechannisms of $\mu\Gamma$CMC is employed only to computing complex structures, such as lists and partial functions.

$\mu\Gamma$CMC was used to implement the back-end of a Haskell compiler. In order to reduce the implementation efforts the front-end of an existent Haskell compiler was used. In particular, the front-end of the Glasgow Haskell Compiler (GHC)[14] was chosen. It translates Haskell code into STG code. As can be seen in the Figure 2, STG is the interface between GHC and the back-end based on $\mu\Gamma$CMC machine.

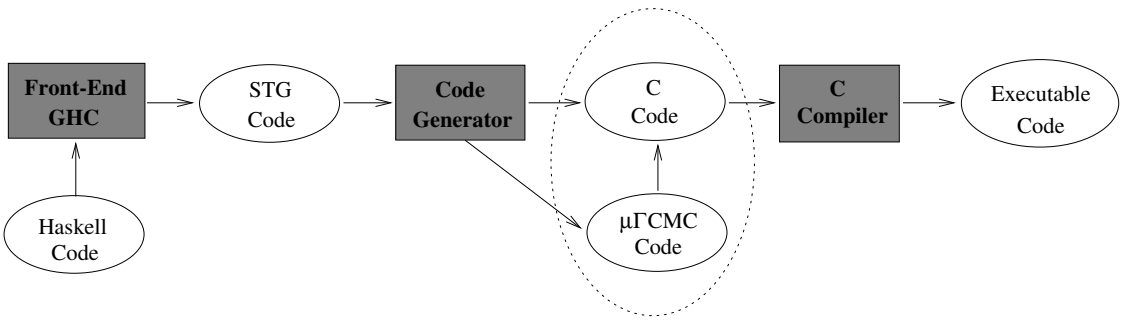


Fig. 2. Structure of *Haskell*/$\mu\Gamma$CMC compiler

3 The Action Semantics Specification

In this section, we describe the action semantics specification for STG language. This specification is composed by three parts: *abstract syntax*, *semantic entities*

and *semantics functions*. In this paper, our focus is on describing how some characteristics judged more relevant in STG were specified using action notation. As a consequence, some less relevant parts of the specification are not shown in this paper. If you would like to see the detailed specification, point to `www.cin.ufpe.br/~fhcj/stgas.tar.gz`. The characteristics are:

- Bindings;
- Higher order functions;
- Lazy evaluation;
- Algebraic Values;
- Pattern Matching.

First, we will show the complete abstract syntax of STG and discuss its most relevant aspects. After this discussion there will be five sections talking about the characteristics listed above.

3.1 Abstract Syntax

In Fig. 3, we present the abstract syntax of STG. A STG program is a collection of *bindings*. One of them is the value of the program and is denoted by *main*. The bindings declared at the top-level are called *globals* while the others, declared in `let` or `letrec` expressions, are called *locals*. Each binding associates an identifier to a *λ-form*, which is composed by a list of free variables, an update flag, a list of arguments and an expression, which denotes, when applied, the value of the λ-form based on the actual values of the formal arguments. As we said earlier, the free variables and the update flag do not have denotational meaning, only operational. Their main purpose is to model construction and update of closures in STG-machine. We will ignore them in the semantic specification.

Note that there is not *nesting of expressions*. As we discussed in the previous section, all function and constructor arguments are simple variables or constants. Programmers should use `let` bindings to build more complex forms of expressions. This characteristic makes STG programs very unreadable.

The `let` expressions build local bindings that will be in the scope of an expression. The difference from `letrec` expressions is that the latter supports mutual recursion among bindings. The `case` expressions make possible pattern matching. The *selector expression* inside a `case` expression have to be evaluated strictly and the given value is used in choosing the appropriate `case` alternative. The value of the `case` expression is then the value of the choosen *alternative expression*. The `default` alternative is applied when there is no alternative to choose. Algebraic values are builded applying a *constructor identifier* to arguments (literals or variables). For differentiating constructor identifiers from function identifiers, the former must have an upper case letter as first letter, while the latter must have a lower case letter in the beginning. The syntax of function and primary operation application and the syntax of literals are straitforward.

The scope rules of STG are static. The bindings accessed by an expression are the globals and the locals defined in the `let` or `letrec` expressions where it is defined.

needs: Data Notation/Characters/ASCII (letter,digit,graphic-character). closed

grammar:

• Programs

(1) Program = ⟦ Binding^{+} ⟧.

• Bindings

(2) Binding = ⟦ Var "=" λ-Form ⟧.

(3) λ-Form = ⟦ "{" Vars$^{?}$ "}" "\"π "{" Vars$^{?}$ "}" "→ " Expression ⟧ .

(4) π = "u" | "n" .

• Expressions

(5) Expression = ⟦ "`let`" Binding^{+} "`in`" Expression ⟧ |
⟦ "`letrec`" Binding^{+} "`in`" Expression ⟧ |
⟦ "`case`" Expression "`of`" Alternatives ⟧ |
⟦ Var "{" Atoms$^{?}$ "}"⟧ |
⟦ Constructor "{" Atoms$^{?}$ "}"⟧ |
⟦ Prim "{" Atoms "}"⟧ |
⟦ Literal ⟧ .

• Case Alternatives

(6) Alternatives = ⟦ Primitive-alternative* Default-alternative ⟧ | ⟦ Algebraic-alternative* Default-alternative ⟧ .

(7) Primitive-alternative = ⟦ Literal "→ " Expression ⟧.

(8) Algebraic-alternative = ⟦ Constructor "{" Vars$^{?}$ "}" "→ " Expression ⟧.

(9) Default-alternative = ⟦Var "→ " Expression ⟧ | ⟦"`default` → " Expression ⟧ .

• Vars

(10) Vars = ⟨ Var ⟨ "," Var⟩* ⟩.

(11) Var = ⟦ lowercase letter ⟨ letter | digit ⟩* ⟧ .

• Constructors

(12) Constructor = ⟦ uppercase letter ⟨ letter | digit ⟩* ⟧ .

• Primary Functions

(13) Prim = "+" | "−" | "*" | "/" | "<" | ">" | "≤" | "≥" | "==" | "!=" | "&" | "|" | "!" .

• Atoms

(14) Atoms = ⟨ Atom ⟨ "," Atom⟩* ⟩.

(15) Atom = ⟦ Var ⟧ | ⟦ Literal ⟧.

• Literals

(16) Literal = ⟦ Numeric-literal ⟧ | ⟦ Character-literal ⟧.

(17) Numeric-literal = ⟦ digit^{+} ⟧ | ⟦ digit^{+} '.' digit^{+} ⟧ .

(18) Character-literal = ⟦ ''' graphic-character ''' ⟧ .

Fig. 3. Abstract Syntax

3.2 Bindings

In STG, functions are declared as λ-forms. To allow refering them in the program code, λ-forms are associated to names (identifiers). These associations are called bindings. A Binding can be *global*, when declared at the top level of the source code, or *local*, when declared in a `let` or `letrec` expression. We assume that an expression, when evaluated, has access to the global bindings and to the local bindings declared in `let` or `letrec` expressions that contain it. The difference between `let` and `letrec` expressions is that in the latter mutual recursion is allowed. *Global bindings* are elaborated by the semantic function **run**, shown below.

introduces: run _

- run _ :: Program → action [giving value].

(1) run ⟦B:Bind$^+$⟧ =
    | elaborate-bind-rec B
    before
    | | evaluate ⟦ "main { }" ⟧
    | then
    | | | enact the thunk-abstraction of the given thunk.
    | | or
    | | | give the given function

Note that **run** elaborates *global bindings* allowing mutual recursion among them, and then evaluates the identifier *main*, which must be bound to a parameterless λ-form which returns the value of the STG program when applied. This is the semantic of the initialization of a STG program.

The equations of the semantic function **evaluate** for `let` and `letrec` expressions were defined as follows.

(1) evaluate ⟦ "`let`" B:Bind$^+$ "`in`" E:Expression ⟧ =
    | elaborate-bind B
    before
    | evaluate E .

(2) evaluate ⟦ "`letrec`" B:Bind$^+$ "`in`" E:Expression ⟧ =
    | make-indirection B
    before
    | | elaborate-bind-rec B
    | before
    | | evaluate E .

Essentially, each one in a set of binding declarations is elaborated by the semantic function **elaborate-bind** or **elaborate-bind-rec** and the expression is evaluated using the elaborated bindings and the other bindings already in scope. For `letrec` expressions, indirections are created for allowing mutual recursion. The equations for semantic-function **make-indirection** are shown next.

- make-indirection _ :: Bind$^+$ → action [binding].

(1) make-indirection ⟨ B_1:Bind B_2:Bind$^+$ ⟩ =
    |make-indirection B_1
    before
    |make-indirection B_2 .

(2) make-indirection ⟦ I:Var "=" L:λ-Form ⟧ = indirectly bind I to unknown.

The semantic function **make-indirection** only creates indirections for each binding, which will be redirected by the semantic function **elaborate-bind-rec**. This semantic function and the semantic function **elaborate-bind** were defined as follows.

- elaborate-bind _ :: Bind$^+$ → action [binding].

(1) elaborate-bind ⟦ I:Var "=" L:λ-Form ⟧ =
    |elaborate-function L
    then
    |recursively bind token of I to the given unevaluated-value.

(2) elaborate-bind ⟨ B_1:Bind B_2:Bind$^+$ ⟩ =
    |elaborate-bind B_1
    before
    |elaborate-bind B_2 .

- elaborate-bind-rec _ :: Bind$^+$ → action [binding].

(3) elaborate-bind-rec ⟦ I:Var "=" L:λ-Form ⟧ =
    |elaborate-function L
    then
    |redirect token of I to the given unevaluated-value.

(4) elaborate-bind-rec ⟨ B_1:Bind B_2:Bind$^+$ ⟩ =
    |elaborate-bind-rec B_1
    and
    |elaborate-bind-rec B_2 .

For each binding declaration, **elaborate-bind** and **elaborate-bind-rec** call the semantic function **elaborate-function** for elaborating an action representation for a function (λ-form), using the reflective facet of action notation (*abstractions*). We will discuss this representation and the definition of **elaborate-function** in the next section (3.3). The identifier of the binding declaration is bound to the *abstraction* given by **elaborate-function**.

3.3 Higher Order Functions

In a programming language, if functions are treated as first class values, allowing them to be stored in data structures, passed as arguments, and returned as a result of evaluation of another function, they are referred as *higher order functions* [9].

Since λ-calculus, the mathematical foundation of functional languages, the support for higher order functions is one of the most intrinsec characteristics

of these languages and, as a consequence, their desirable properties to the programming practice have been extensively studied. Higher order functions are a powerful tool for increasing *modularity* and *abstraction* of programs.

Except for primitive functions, which must be always saturated, STG provides support to higher order functions. For that, in our specification, each function is treated as a *curried function*, that is, a function with only one argument that returns, when applied to a value, another curried function, which can be applied to another argument, or a simple value. Thus, the application of a function f, with three arguments, can be seen as follows.

$$f\{x, y, z\} \sim ((f\{x\})\{y\})\{z\}$$

The application of the function f can be interpreted as the application of f to the first argument, returning another function that is applied to the second argument, returning another function that is finally applied to the third argument, returning a simple value. In fact, the semantics of application of functions in STG coincides with the semantics of application of curried functions, unlike its syntax suggests and in spite of that STG-machine, in a function application, passes actual arguments all at a time. The support for *partial application of functions* allows functions being passed as arguments to other functions.

In the semantic entities definition, we defined a sort, called **function**, for encapsulating an abstraction that is an action representation of a function, the *function abstraction*.

introduces: function, function of _.

(1) a:action ⇒ function of abstraction of a :function.
(2) f = function of a ⇒ the function-abstraction of f:function = a.

A *function abstraction* is elaborated by the semantic function **elaborate-function**. It receives a λ-form and builds the correspondent function abstraction, which is bound to the appropriate identifier by **elaborate-bind** or **elaborate-bind-rec** semantic functions, shown later in this paper.

- elaborate-function _ :: λ-Form → action [giving a function].

(1) elaborate-function ⟦ "{" F_v:Vars "}" "\" π "{" ⟨ A_1:Var "," A_n:Vars ⟩ "}" "→ " E:Expression ⟧ =
    give function of closure abstraction of
    ||bind token of A_1 to the given argument
    before
    ||elaborate-function ⟦ "{" F_v "} \" π "{" A_n "}" "→ " E ⟧.
(2) elaborate-function ⟦ "{" F_v:Vars "}" "\" π "{" A:Var "}" "→ " E:Expression ⟧ =
    give function of closure abstraction of
    ||bind token of A to the given argument
    before
    ||evaluate E.

(3) elaborate-function ⟦ "{" F_v:Vars "}" "\" π "{" "}" "→" E:Expression ⟧ = evaluate E.

For a λ-form with more than one argument, note that its function abstraction, when enacted, binds the given actual argument to the name of the first formal argument and then returns another function abstraction that will consume the remaining arguments in the same manner. In the case of a function abstraction having only one argument, the expression is evaluated, using the generated bindings for the arguments, local bindings in scope and global bindings. *Curried functions* are implemented in this way. The following equation for *evaluate* semantic function specifies the correspondent action for a *function application*.

- evaluate _ :: Expression → action [using current bindings][giving an unevaluated-value].

⋮

(1) evaluate ⟦ I:Var "{" A:Atoms "}" ⟧ =
    | give the function bound to the token of I
    | then
    | saturate-function A
    | or
    | give the thunk bound to the token of I

(2) evaluate ⟦ I:Var "{" "}" ⟧ =
    | give the thunk bound to the token of I
    | or
    | give the function bound to the token of I

⋮

In a function application, when more than necessary actual arguments are passed to a function, the remaining actual arguments are ignored. The semantic function **saturate-function** applies the given arguments, one by one, to the function abstraction bounded to the identifier. In the case of less than necessary arguments being passed to a function (*partial application*), the result of the application is a function abstraction which can consume the remaining arguments when applied. The referred semantic function **give-atom** only gives, as a transient, the argument which can be a literal or a "value" bounded to an identifier.

- saturate-function _ :: Atoms → action [giving unevaluated-value].

(1) saturate-function A:Atom =
    | give the given function
    | and then
    | give-atom A
    | then
    | enact the application of the function-abstraction of the given function#1
    | to the given argument#2 .

```
(2)  saturate-function ⟨ A:Atom "," As:Atoms ⟩ =
         |saturate-function A
         then
         ||give the given thunk
         or
         |||give the given function
         |then
         |||saturate-function As .
```

In the specification, the reader can see that an identifier can be bound to a *function* or a *thunk*. Thus, these are the sort of values that can be given as transients by **give-atom** semantic function, when it is applied to an identifier. This explains how functions are passed as arguments to other functions.

3.4 Lazy Evaluation

There are two main groups of functional languages: the first one contains the functional languages with *strict* semantics while the other includes the functional languages that support *non-strict* semantics. Respectively, the most important languages in each group are ML[2] and *Haskell*. A function with a strict semantics have the following property: whenever one of its arguments has an undefined value, its value is undefined too (Fig. 4). This is not necessarily true for non-strict functions. For example, if the value of the undefined argument is not referenced in the body of the function and the other arguments are defined, the function value can be defined. In functional languages, non-strict semantic is guaranteed when *normal order reduction* from λ-calculus is supported while strict semantics is a characteristic of functional languages that support *applicative order reduction*[9]. In modern non-strict functional languages, normal order reduction is implemented by *lazy evaluation mechanism*.

$$f\ \bot = \bot$$

Fig. 4. Strict Semantics Property

In the lazy evaluation mechanism, the arguments of a function are passed unevaluated. They are evaluated only when their values are strictly necessary (Fig. 5 show a simple example of lazy evaluation using a syntax of λ-calculus). Naively implemented, this approach can lead to low performance in respect to time and space [10]. Fortunately, the recent development in technology for compiling lazy functional languages[12], like *haskell*, has shown that *lazy evaluation* can be as efficient as *eager evaluation*, the technique in which arguments are evaluated before being applied to the function (mechanism of implementation of applicative order reduction). In fact, some applications execute faster when implemented with lazy functional languages[10]. Benchmarks have shown that

actual compiled functional language code can run fastest as compiled C and Fortran code for important practical applications [8].

$$(\lambda x \lambda y \rightarrow \times\ x\ y)(\times\ 2\ 6)(+\ 3\ 2)$$

$\Rightarrow\ \times\ (\times\ 2\ 6)\ (+2\ 3)$	$\Rightarrow\ (\lambda x \lambda y \rightarrow \times\ x\ y)\ (\times\ 2\ 6)\ 5$
$\Rightarrow\ \times\ (\times\ 2\ 6)\ 5$	$\Rightarrow\ (\lambda x \lambda y \rightarrow \times\ x\ y)\ 12\ 5$
$\Rightarrow\ \times\ 12\ 5$	$\Rightarrow\ \times\ 12\ 5$
$\Rightarrow\ 60$	$\Rightarrow\ 60$
Lazy Evaluation (Normal)	*Eager Evaluation (Applicative)*

$$(\lambda x \lambda y \lambda z \rightarrow +\ x\ y)(+\ 2\ 20)\ 4\ (/\ 1\ 0)$$

$\Rightarrow\ +\ (+\ 2\ 20)\ 4$	$\Rightarrow\ (\lambda x \lambda y \rightarrow \times\ x\ y)\ (+\ 2\ 20) 4\ \bot$
$\Rightarrow\ +\ 40\ 4$	$\Rightarrow \bot$
$\Rightarrow\ 80$	
Lazy Evaluation (Normal)	*Eager Evaluation (Applicative)*

Fig. 5. Two Examples of Evaluation using *Lazy* and *Eager* Mechanisms

Many researchers have argued that lazy functional languages are more powerful and general than its eager counterparts because they have some desirable properties that influences the programming practice[9, 3, 10]:

- independence of the program semantics from the evaluation mechanism;
- support for representation of infinity data structures;
- high degree of *modularity*.

Because STG was originally developed to be the "abstract machine code" for an abstract machine that supports implementation of non-strict (lazy) functional languages (STG-machine), it is natural that STG is non-strict too. In this action semantics specification, we provide support for unevaluated expressions (*thunks*). The **evaluate** semantic function, when applied to an expression always returns a *thunk* or a *function*, as discussed before. A thunk (unevaluated expression) is a sort that, like function, encapsulates an abstraction (*thunk abstraction*), which returns a value when enacted. A thunk is only enacted when its value is strictly necessary. Following, we show the specification for thunks contained in the semantic entities definition.

introduces: thunk, thunk of _.

⋮

(1) a:action a ⇒ thunk of abstraction of a :thunk.
(2) t = thunk of a ⇒ the thunk-abstraction of t:thunk = a.
(3) unevaluated-value = thunk | function. (*disjoint*)
(4) argument = unevaluated-value.

Note the similarity between the definitions of the sorts *thunk* and *function*. The sort *unevaluated-value* contains individuals that belongs to the sorts *thunk* or *function*. The same can be said about the sort *argument*.

As we presented in the previous section, the semantic function **run** elaborates the global bindings and evaluates the function *main*, which have access to the elaborated global bindings. This evaluation can return a *thunk*, which can return a value when its thunk abstraction is enacted, or a function. Another situation where a thunk must be enacted is when it is applied to primary operations. The general form of the evaluation of a primary operator application is described below.

```
•   evaluate _ :: Expression → action [using current bindings][giving an unevaluated-value].
    ⋮
(1) evaluate ⟦ P:Prim "{" A:Atoms "}" ⟧ =
        |give-arguments A
        then
        |apply-operator P   .
    ⋮
•   apply-operator _ :: Prim → action [using given data][giving a thunk].
    ⋮
(2) apply-operator ⟦ bin_oper ⟧ =
        give the application of the thunk of
        |abstraction of
        ||||enact the thunk-abstraction of the given thunk#1
        |||and
        |||enact the thunk-abstraction of the given thunk#2
        ||then
        |||give the action_bin_oper(the given number-value#1,
        |||      the given number-value#2)
        |to the given data
    ⋮
```

In this **evaluate** semantic function equation, all the arguments are given at the same time to the **apply-operator** semantic function. The *bin_oper* argument is a primitive operation while *action_bin_oper* is its correspondent action. For example, *sum* is the action correspondent to the primary operation "+". In the returned abstraction, the thunks of the actual arguments of the primary operation are enacted and the values are used for calculating the result of the operation.

Later in this paper, in Sect. 3.6, when discussing about *pattern matching*, we will see that the third point where a thunk is enacted is when scrutinizing

the *selector value* of a *case expression*, by evaluating its *selector expression* and enacting the given thunk.

We pointed to the three points in the specification where thunks must be enacted. In all other cases, values are kept unevaluated in thunk abstractions, leading to a lazy evaluation schema.

3.5 Algebraic Values

Algebraic data types (ADT's) can help programmers to increase data abstraction, whether or not the language is functional. Data abstraction, a disseminated concept of programming languages in general, improves the *modularity*, *security*, and *clarity* of programs[9].

In *Haskell*, an ADT is defined through a **data** declaration. For example, following we show the definition of an ADT that represents a *tree* of values of type *t*.

```
data Tree t = Nil |
              Node t (Tree t) (Tree t)
```

In Fig. 6, we show a binary tree of integers and its representation as a *Tree Int* ADT value. See that values are builded by the application of a constructor (**Node** or **Nil** in this case) to component values of a required type.

In STG, there is not a way of declaring ADT's, like **data** declarations in *Haskell*, but algebraic values can be builded by the application of constructors to literals or identifiers. In fact, in STG, there is no need for a way of defining an ADT, because it is proposed to be an intermediate "low level" language and, in general, type checking and inference should be performed before the translation of the high level code to it. This is the strategy adopted by GHC and indirectly adopted by *Haskell*/$\mu\Gamma$CMC compiler, which uses the STG code generated by GHC. The application of a constructor to its components must be always saturated (There is not partial application for constructors) and the components can be scrutinized by pattern matching (*case expressions*).

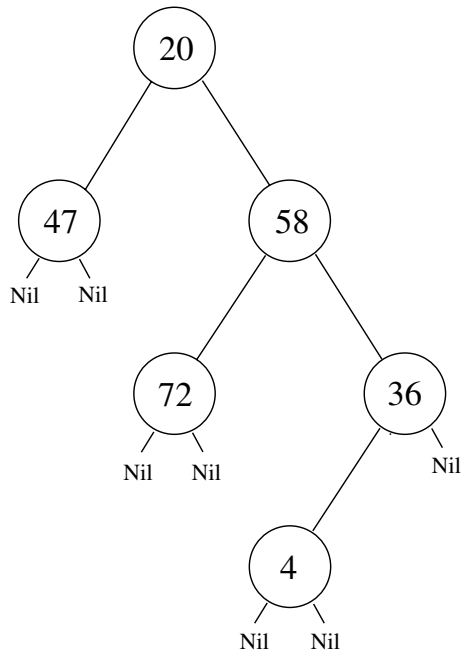
In the semantic specification, the evaluation of an application of a constructor to its components (construction of an algebraic value), gives a thunk which gives an individual of the sort *algebraic-value* when enacted. This sort is defined as follows in the semantic entities definition.

introduces: primitive-value , algebraic-value, value.

⋮

(1) algebraic-value = (string, unevaluated-value*).

An algebraic value is represented in action notation as a tuple, where the first component is the name of the constructor and the others are its components. The components are stored as *unevaluated values* (*thunks* or *functions*) until their values be strictly necessary (*lazy evaluation*). The equations of the



Node 20 (Node 47 Nil Nil) (Node 58 (Node 72 Nil Nil) (Node 36 (Node 4 Nil Nil) Nil))

Fig. 6. Tree of Integers and its representation through an algebraic value

semantic function **evaluate** for application of constructors and the equations of the semantic function **saturate-constructor** are presented below.

```
(1)  evaluate ⟦ C:Constructor "{" A:Atoms "}" ⟧ =
         |give (token of C)
         then
         |saturate-constructor A .

(2)  evaluate ⟦ C:Constructor "{" "}" ⟧ =
         give the thunk of the provision of the (token of C).

     ⋮

•    saturate-constructor _ :: Atoms → action [giving thunk].

(3)  saturate-constructor A:Atom =
         ||give-atom A
         |and
         ||give the given algebraic-value
         then
         |give (the given algebraic-value, the given unevaluated-value)
         then
         |give the thunk of provision of the given algebraic-value.
```

```
(4)  saturate-constructor ⟨ A:Atom "," As:Atoms ⟩ =
         ||give-atom A
         and
         ||give the given algebraic-value
         then
         |give (the given algebraic-value, the given unevaluated-value)
         then
         |saturate-constructor As .
```

The semantic-function named **saturate-constructor** builds the tuple that corresponds to the algebraic value. Note that when the constructor has no components, only the name of the constructor is given as transient. In the next section, when talking about pattern matching we will see how components of an algebraic value are scrutinized.

3.6 Pattern Matching

Equational reasoning is one of the most important and strongly encouraged programming methodology in the design and construction of programs. It is supported intrinsecly by functional languages, which have no side effects[9, 4]. In general, in these languages, equations are part of the syntax and *pattern matching* goes along with that, allowing write several equations for defining the same function, only one of which should be applicable in a given situation.

The mechanism of pattern matching of STG, only by case expressions, makes STG programs very difficult to understand, but it is general, in the sense that pattern matching can be seing as the primitive behavior of a case expression. The general form of a case expression is:

$$
\begin{array}{l}
\textbf{case } e \textbf{ of} \\
\quad pat_1 \rightarrow e_1; \\
\quad pat_2 \rightarrow e_2; \\
\quad\quad \vdots \\
\quad pat_n \rightarrow e_n; \\
\quad id \quad\; \rightarrow e_{defalut};
\end{array}
$$

The guard expression e is strictly evaluated. Then, the value scrutinized should or not match one of the patterns (pat_i $i = 1, \ldots, n$). If a pattern pat_i matches the scrutinized value then the value of the case expression is the value of e_i. If no patterns match the value, it is bounded to the identifier id and the value of the case expression is $e_{default}$. There are two kinds of patterns: for *primitive values* and for *algebraic values*. The former corresponds to a value while the latter has the form $C\{v_1, \ldots, v_n\}$, where C is a constructor and v_i is an identifier. The components of the algebraic value are bounded to the identifiers of the pattern that have the same constructor as the scrutinized value constructor. Following we show the **evaluate** semantic function equation for *case expressions*.

(1) evaluate ⟦ "case" E:Expression "of" A:Alternatives ⟧ =
    |evaluate E
    then
    ||enact the thunk-abstraction of the given thunk
    then
    |||exhaust-alternatives A .
    |trap
    |||give the given thunk.

The *selector expression* e is evaluated and its thunk is enacted. The semantic function **exhaust-alternatives**, presented below, try to discover which pattern (pat_i) matches the value and returns the thunk resulted from evaluation of the correspondent expression (e_i). Following we also show the semantic functions **accept-primitive-alternative**, **accept-algebraic-alternative**, **accept-default-alternative**, and **bind-vars-to-components**, used in testing each case alternative and in choosing the appropriate one based on the scrutinized value in the case selector. The first semantic function is for testing primitive alternatives and the second is for algebraic ones. The third applies the default alternative when none of the primitive or algebraic alternatives were choosen. The fourth only binds the name identifiers from algebraic guards to its correspondent values in a scrutinized and choosen algebraic value. It is used by **accept-algebraic-alternative** semantic function.

- exhaust-alternatives _ :: Alternatives → action [giving thunk].

(1) exhaust-alternatives ⟦ P:Primitive-alternative$^+$ D:Default-alternative ⟧ =
    |accept-primitive-alternative P
    and then
    |accept-default-alternative D .
(2) exhaust-alternatives ⟦ A:Algebraic-alternative$^+$ D:Default-alternative ⟧ =
    |accept-algebraic-alternative A
    and then
    |accept-default-alternative D .
(3) exhaust-alternatives ⟦ D:Default-alternative ⟧ = accept-default-alternative D .

- accept-primitive-alternative _ :: Primitive-alternative$^+$ → action [giving thunk].

(1) accept-primitive-alternative ⟦ L:Literal "→ " E:Expression ⟧ =
    |check (value of L is the given value)
    and then
    ||evaluate E
    then
    ||escape with the given thunk.
(2) accept-primitive-alternative ⟨ P_1:Primitive-alternative P_2:Primitive-alternative$^+$ ⟩ =
    |accept-primitive-alternative P_1
    and then
    |accept-primitive-alternative P_2.

- accept-algebraic-alternative _ :: Algebraic-alternative$^+$ → action.

(1) accept-algebraic-alternative ⟦ C:Constructor "{" I:Vars "}" "→ " E:Expression ⟧ =
    check (value of C is the first of the given algebraic-value)
    and then
        give the rest of the given algebraic-value
        then
            bind-vars-to-components I
            then
                evaluate E
                then
                    escape with the given thunk.

(2) accept-algebraic-alternative ⟦ C:Constructor "{" "}" "→ " E:Expression ⟧ =
    check (value of C is the first of the given algebraic-value)
    and then
        evaluate E
        then
        escape with the given thunk.

(3) accept-algebraic-alternative ⟨ P_1:Algebraic-alternative P_2:Algebraic-alternative$^+$ ⟩ =
    accept-algebraic-alternative P_1
    and then
    accept-algebraic-alternative P_2.

- accept-default-alternative _ :: Default-alternative → action [giving thunk].

(1) accept-default-alternative ⟦I:Var "→ " E:Expression ⟧ =
    bind token of I to the thunk of the provision of the given data
    before
        evaluate E
        then
        escape with the given thunk.

(2) accept-default-alternative ⟦"`default` → " E:Expression ⟧ =
    furthermore
        evaluate E
        then
        escape with the given thunk.

- bind-vars-to-components _ :: Vars → action.

(1) bind-vars-to-components I:Var =
    bind token of I to the first of the given tuple.

(2) bind-vars-to-components ⟨ I:Var "," Is:Vars ⟩ =
        bind-vars-to-components I
        and
        give the rest of the given tuple
    then before
    bind-vars-to-components Is.

4 Conclusions and Lines for Future Works

In this paper, we presented an action semantics for STG, a pure non-strict higher order functional language intended to be an intermediate code for compiling process of higher level functional languages. In spite of the close relashionship between STG and STG-machine, a virtual machine that supports implementation of functional languages, our main goal was to show the meaning of STG programs independently from how they execute on STG machine, showing how some important aspects of a non-strict higher order functional language can be specified using action semantic, like *bindings*, *higher order functions*, *lazy evaluation* (*non-strict semantics*), *algebraic values* and *pattern matching*.

Action semantics was a very useful tool in the specification of STG semantics. It gave us a very modular and easy to understand and extent documentation and could allow us to generate a compiler for STG from the specification. With that, we could use some of the compiler generator systems available[1]. This is a very useful feature of action semantic formalism that can be used for evaluation, using a compiler prototype, of semantic and syntax aspects of new developed languages. For this purpose, we intend to use the compiler generator ABACO[6]. Besides, due to the high degree of modularity provided by action notation, this work can be seen as an intial step for specification of "higher level" functional languages, like *Haskell*. Also, we think that, due to its clear understanding, action notation is a very useful tool for teaching purposes, allowing programming languages students in understanding the meaning of the most relevant aspects of programming languages and paradigms.

References

1. Macedo A. C. B. , Moura, H. P.: Investigating Compiler Generator Systems. Proceedings of the 4th Brazilian Symposium on Programming Languages (2000), 259-265
2. Wikström, A.: Standard ML. Prentice-Hall, Englewood Cliffs, New york, 1988.
3. Hughes, J.: Why Functional Programming Matters. The Computer Journal (1989), Vol.32(2).
4. Hammond, K., Michaelson, G.: Research Directions in Parallel Functional Programming. Springer Verlag (1999)
5. Kyung-Goo-Doh. Action Semantics: A Tool for Developing Programming Languages. Technical Report 93-1-005, The University of Aizu (1993)
6. Meneses, L. C. S.: Uso de Orientação o a Objetos na Prototipação de Semântica de Ações. Master's Thesis, Federal University of Pernambuco (1998)
7. Mosses, P. D.: Action Semantics. Cambridge Tracts in Theoretical Computer Science, Vol. 26, Department of Computing Science, University of Glasgow (1992)
8. Hartel, P. H. , Alt, M. , Beemster, W., Lins, R. D., *et al*: Benchmarking Implementation with "PseudoKnot", a Float Intensive Benchmark. Journal of Functional Programming (1996)
9. Hudak, P.: Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys (1989), Vol. 21(3-4), 359-411

10. Bird, R., Jone, G., and de Moor, O.: More Haste, Less Speed: Lazy Versus Eager Evaluation. Journal of Functional Programming, Vol. 7 (1997), 541-547
11. Lima, R. M. F.: Doctor Thesis, Centre of Informatics, UFPE, July 2000.
12. Jonhson, T.: Compiling Lazy Functional Languages. PhD Thesis, Chalmers Tekniska Hgskola, Gteborg, Sweend (1987)
13. Peyton Jones S. L.: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine *Version 2.5*
14. Peyton Jones, S. L., Hall, C., Hammond, K., Partain, W.: The Glasgow Haskell Compiler: a Technical Overview. Proceedings of Joint Framework for Information Technology Technical Conference, Keele (1993) 249-257
15. Peyton Jones, S. L., Hughes, J., *et al*: Report on the Programming Language Haskell, A Non-Strict, Purely Functional Language, Version 1.4 (1997)
16. Peyton Jones, S. L., Launchburry, J.: Unboxed Values as First Classes Citizens in a Non-Strict Functional Language. Proceedings of the 1991 Conference of Functional Programming Languages and Computer Architecture, Cambridge (1991)
17. Thompson, S., Lins, R. D.: The Categorical Multi-Combinator Machine: CMCM. The Computer Journal (1992), Vol.3, P. 2, 170-176

A Formal Description of SNMPv3 Standard Applications using Action Semantics

Diógenes Cogo Furlan, Martín A. Musicante, and Elias Procópio Duarte Jr.

Federal University of Paraná. Dept. Informatics
P.O.Box 19081 - CEP 81531-990 - Curitiba PR - Brazil
{diogenes,mam,elias}@inf.ufpr.br

Abstract. The Simple Network Management Protocol version 3 (SNMPv3) is the Internet standard management architecture. A system based on SNMPv3 is composed of management entities which communicate using the management protocol. Entities are composed of an *engine*, and a number of standard *applications*. An entity uses standard applications to communicate using SNMP. These applications include command and notification generators and responders. Applications use the services of the engine to send and receive messages. The specification of SNMPv3 is given informally in a series of IETF documents. This work presents a formal description of SNMPv3 standard applications using Action Semantics. The main goals of our formal description are to enhance the understandability of standard applications, as well as to establish the foundation for the automatic generation of SNMP entities implementations in the future.

1 Introduction

As networks become larger and more complex the need for effective network management systems becomes critical. The purpose of these systems is to help human managers in the tasks of monitoring and controlling computer networks. The Simple Network Management Protocol version 3 (SNMPv3) is the Internet standard management architecture. An SNMPv3 system is composed of management *entities* which communicate using the management protocol [15, 2]. SNMPv3 entities have traditionally been called managers and agents. Managed nodes contain an *agent*, i.e. a management entity which have access to management instrumentation. Each system has at least one Network Management Station, which runs at least one manager entity. *Managers* are collections of user-level applications, which may aim at performance evaluation or fault diagnosis, among others. There is currently a very large number of SNMP-based systems available, both commercial and on the public-domain.

SNMP management entities (figure 1) are composed of an *engine*, and a number of standard *applications*. An entity uses standard applications to communicate using SNMP. These applications include the command generator, the notification generator, the notification generator and the notification responder, which are all formally specified in this work. This is seen as the first step towards

fully defining whole management entities, and allowing the automatic generation of MIB implementations from specifications.

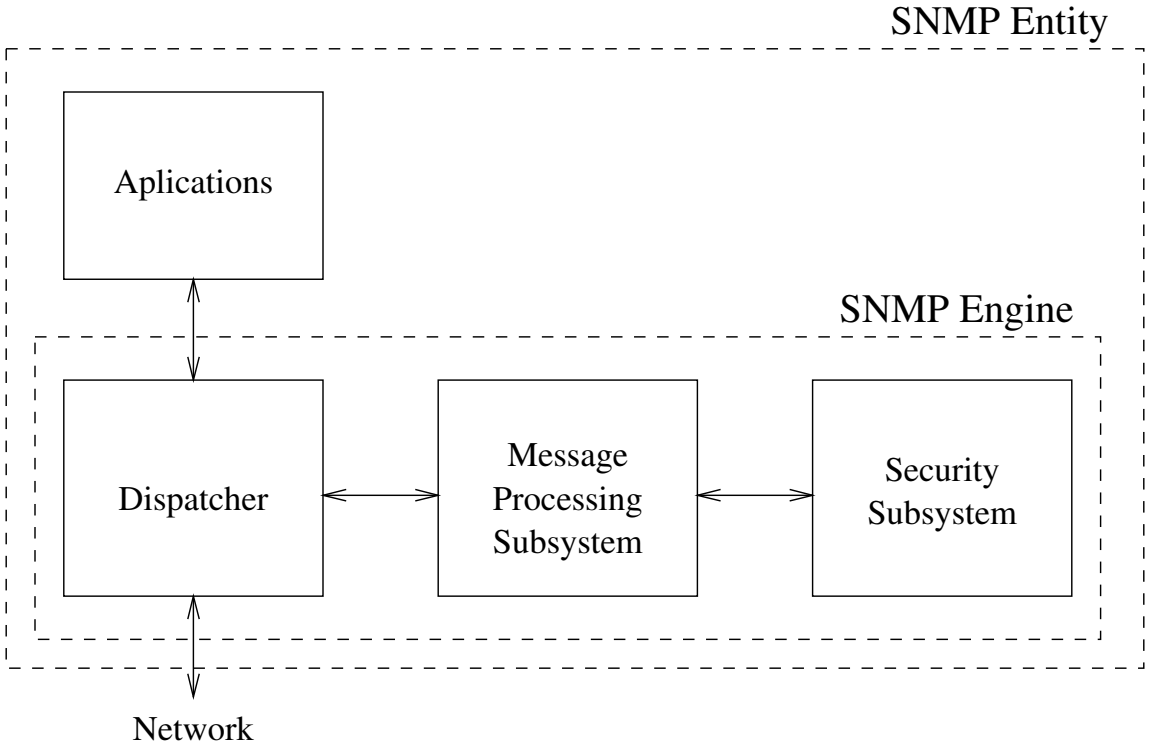


Fig. 1. SNMPv3 Entity.

Currently the semantics of SNMP components is give informally in [4, 2]. There is informal English text explaining each object's behavior. These informal descriptions are usually vague and incomplete. They are open to misinterpretation and may lead to inconsistent implementations. In this work we use *Action Semantics* [10] to formally describe the semantics of SNMPv3 standard applications. Action Semantics is a formal framework for semantic description, developed to provide readable descriptions of real-life languages. Action semantic descriptions map abstract syntax to semantic entities, which are defined inductively using semantic equations. The semantic entities employed are *actions* rather than higher-order functions, used in other formalism, and the essence of actions is much more *computational* than that of pure mathematical functions.

In [5] Action Semantics was used to define the semantics of an specific management entity, the Routing Proxy [6].

The rest of the paper is organized as follows. Section 2 reviews the standard SNMPv3 architecture, concentrating on the standard applications and how their role within a management entity. Section 3 presents an introduction to Action Semantics, the notation we use to formally define the standard applications. Section 4 is the formal specification of the command and notification generators and responders. Section 5 contains the conclusions.

2 SNMPv3 Standard Applications

The Simple Network Management Protocol version 3 (SNMPv3) is the Internet standard management architecture [4]. An SNMPv3 system is composed of nodes running management entities which communicate using the management protocol. Each system has at least one Network Management Station, which runs at least one manager entity. Managed nodes contain an agent, i.e. a management entity which have access to management instrumentation. There are other kinds of management entities, like proxies [4].

Each SNMPv3 entity can behave as an agent as well as as a manager [2]. An SMNP entity is acting as a manager when it initiates a management operation (i.e. when it generates requests for the operations *get*, *getNext*, *getBulk*, *set*; or *inform*) or when it answers to notifications. A SNMP entity is acting as an agent when it responds to management operations and when it sends a notification. The same SNMP entity can act as a manager and as an agent.

The management protocol is used to communicate management information among SNMP entities. Management information can be seen as a collection of objects, organized in Management Information Bases (MIB's). The abstract syntax of each MIB is defined using a subset of the Abstract Syntax Notation 1 (ASN.1) language [3]. Messages exchanged among entities contain ***Protocol Data Units*** (PDU's) defined in [1, 2].

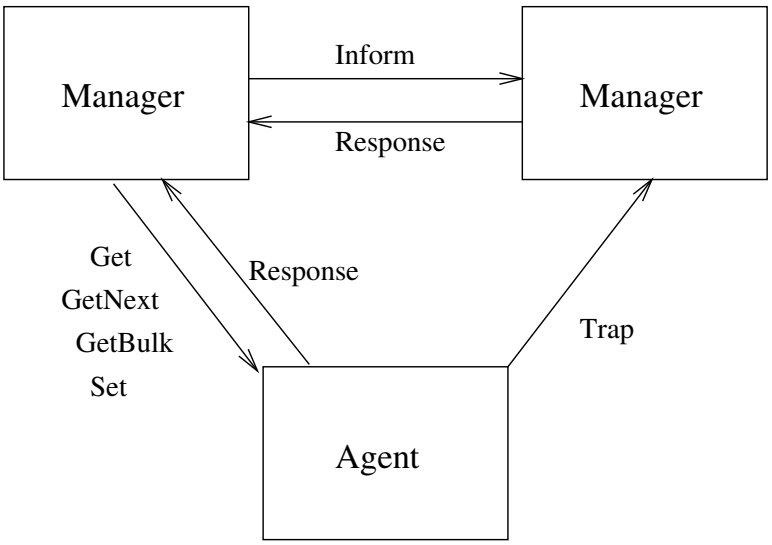


Fig. 2. SNMPv3 Interactions.

There are three types of interactions among SNMPv3 entities, as shown in figure 2. The first one is the request-response, sent when an entity acting as manager sends a requirement to an entity acting as agent. This kind of interaction is used to retrieve or update management information which is associated to the

managed device. The second type of interaction is a request-response between managers. This is called a notification among managers.

The third type of interaction between entities occurs when an entity acting as an agent sends an alarm *trap* to a manager. This type of interaction does not employ acknowledgements, and is used to notify the manager that an exceptional situation has been detected.

The architecture of a SNMP entity is described in figure 1 [4]. It is composed of a number of standard ***SNMP applications***, which define the functionality of that entity. An ***SNMP engine***, which provides services and support to the applications, such as lower level communication with other entities, including cryptography, authentication and control over the managed objects.

In this work we consider that standard applications communicate which each other by sending and receiving PDU's. In an actual implementation, the communication is performed at the engine level: SNMP engines receive the PDU's from their applications, encoding them into messages which will be sent to adequate entities.

In the next section, the following four standard SNMP applications are formally specified [9]:

- ***Command Generators*** are applications which monitor and manipulate management data by sending requests and processing replies.
- ***Command Responders*** are applications that receive and process requests. They also generate replies for the received requests.
- ***Notification Originators*** are applications that generate messages, which are based on particular events or conditions.
- ***Notification Receivers*** are applications that wait for notification messages.

An entity is acting as a manager when it contains applications of the type Command Generator, Notification Originator and Notification Receiver. An entity is acting as an agent when it contains applications of the type Command Responder and Notification Originator.

3 Action Semantics

Action semantics [10] is a formal framework for the specification of programming concepts. Action Semantics was developed to enhance the readability of descriptions; it uses English words, in a way that formal descriptions can be understood by readers not used to with mathematical notation. A number of real-life languages and applications has been described using action semantics, among them, we can cite [13, 14, 8, 12].

Action semantics uses a special notation to describe actions. This notation is called ***Action Notation***, and it is used in action semantic descriptions very much in the same way as the λ-notation is used in denotational semantics [16].

The symbols used in action notation are intentionally verbose, so that English-like phrases can be used -completely formally- to express most of the concepts present in computing.

In Action Semantics, the meaning of each phrase of a language is represented by special entities called *actions*. Actions can be performed, yielding various possible outcomes: normal termination (complete), exceptional termination (escape), unsuccessful termination (fail) or non-termination (diverge). Action notation provides some ***primitive actions***, and various ***combinators*** for forming complex actions, corresponding to the main fundamental concepts of programming languages.

A ***data notation*** is used to describe the information processed by actions. The standard data notation (included in action notation) provides a collection of algebraically defined abstract data types, including numbers, characters, strings, sets, tuples, maps, etc.; further data may be specified *ad hoc*.

There is also a third class of entities in action notation, called ***yielders***. A yielder is an expression representing unevaluated data, whose value depends on the current information available to the primitive action in which it occurs. Yielders are evaluated to yield data. An example of a standard yielder is the value stored in the given cell, which depends on the current contents of the storage and the information passed to the action in which the yielder appears.

The behavior of each primitive action and action combinator of Action Notation can be classified according to the following facets:

Basic: This facet deals exclusively with flow of control. The control behaviors contained in this facet include sequential and interleaved composition of actions, as well as bounded non-deterministic. For example, the basic facet of the action combinator "A_1 and A_2" specifies that its left and right subactions are performed in interleaving. The behavior of the action "A_1 and then A_2" is such that A_1 must be finished before A_2 is started. The action "A_1 or A_2" represents a choice between actions; it can be used to describe **if-then-else** control structures.

Functional: This facet deals with transient data, which is given to or received by an action. For example, when the basic action "give the given natural" receives a natural number as transient data, it completes and gives the received natural number as a transient. The compound action "A_1 then A_2" performs action A_1 first, and the transient data produced by A_1 are supplied to A_2, which is performed after A_1 completes.

Declarative: This facet deals with the manipulation of scoped information, represented by associations of *tokens* to *bindable data*. For example, the basic action "bind "max-length" to 256" completes its performance, producing a binding that maps the token "max-length" to the natural number 256.

Imperative: Storage handling primitives are provided by this facet. A store, in the action notation context, is a mapping from *cells* to storable data. For example, the action

```
|allocate a cell
then
|store 26 in the given cell
```

allocates a new cell of the store, and stores a value 26 in it. This action combines features of both the functional and imperative facets. (Vertical bars are used to guide the correct association of actions, as an alternative to parentheses.)

Communicative: This facet provides a system of *agents*, whose task is to perform actions. Agents can communicate using asynchronous message passing. Each agent has its own *communication buffer*, in which all the messages sent to the agent are placed. Communication is reliable, in the sense that no message is ever lost during transmission; however, there is no bound to the time taken for a message to arrive to its destination buffer after transmission begins. Each agent is created with its own store.

Encapsulation of actions as data is also provided within Action Semantics. This feature gives a simple way to support the description of procedure and function abstractions in programming languages. For example, the performance of the action

```
give abstraction of |allocate a cell
                    then
                    |store 26 in the given cell
```

completes, giving an *abstraction* as a transient. An *abstraction* is an item of data which encapsulates an action. Abstractions can be *enacted*; this operation results in the performance of the encapsulated action. Both transients and bindings can be supplied to the abstraction.

For a more detailed description of action notation, the reader can refer to [11] or [17] (the former covers all the aspects related to the formal system; the latter does not cover the communicative facet nor the operational semantics of the notation).

4 SNMPv3 Entities

The purpose of this work is to specify the standard applications within a SNMPv3 entity. Four applications are specified: the core of the Command Responder and the Notification Receiver, as well as primitives for the specification of Command Generators and Notification Originators. The first two applications are responsible for the reception and processing of incoming requests. The latter primitives can be used in the formal description of specific client applications.

The other components of SNMPv3 entities, i.e: the SNMP engine, as defined in [4], are beyond the scope of this work, and are being specified in a companion paper [7].

4.1 Command Responder

The action "CommandResponder" below, represents the Command Responder application. This application awaits for *Get*, *GetNext*, *GetBulk* or *Set* requisitions. The action defines one case for each type of requisition. The "or" combinator represents a choice among actions. Only one of the "service" actions will be chosen, depending on a tag of the received PDU.

- CommandResponder :: action

(1) CommandResponder =
unfolding
||accept a message[from any entity][containing a PDU]
then
||service *Get*
or
||service *GetNext*
or
||service *GetBulk*
or
||service *Set*
then
||unfold

The "CommandResponder" action fetches the communication buffer for a message containing a PDU. One of the services will be chosen, depending on a tag, which is attached to the PDU. The received message is passed to the services as functional information.

The "unfolding ... unfold" construction represents a loop: Each time "unfold" is reached, the enclosing action is performed once more, from the point of the previous, more internal "unfolding" on.

Each of the "service _" actions above corresponds to one SNMPv3 operation, for an entity acting as an agent.

The general definition of a service can be described as follows:

- service _ :: tag → action

(2) service Op:tag =
```
|give the given PDU tagged with Op
then
||producePduBindings
hence
|||process Op
|then
|||give a PDU containing (the integer bound to "request-id",
|||                       the given integer#1, the given integer#2,
|||                       the given VarBindList#3)
|then
|||send a message[to the entity bound to "request-entity"]
|||              [containing the given PDU tagged with Response ]
```

- serviceWithoutResponse _ :: tag → action

(3) serviceWithoutResponse Op:tag =
```
|give the given PDU tagged with Op
then
||producePduBindings
hence
||process Op
```

The action "service _" receives a message as functional information. The primitive action "give _" will complete, in the case that the PDU in it is of the kind specified by Op, and will fail otherwise. The combination this checks and the combinator "or" (in the Command Responder action) achieves the result of choosing *at most* one service for each message received by the application.

After receiving the message, the service must interpret the PDU inside the message. This is done by "producePduBindings", which binds standard names to parts of the contents of the received PDU.

The service is then processed following the informal specification in [2]. These actions are defined ahead in section 4.2.1.

After the received PDU is processed, a reply must be sent back to the entity that requested the service. This is done by assembling the corresponding PDU, which will be sent, within a message, to the requesting entity.

The "serviceWithoutResponse _" action is similar to "service _". The only difference is that no reply is composed. This action will be used by the notification receiver, to process $Trap$s.

4.2 Notification Receiver

The "NotificationReceiver" waits for $Inform$ and $Trap$ requests. Its description is similar to that of the command responder:

- NotificationReceiver :: action

(4) NotificationReceiver =

```
unfolding
  ||accept a message[from any entity][containing a PDU]
  then
  |||service Inform
  |or
  |||serviceWithoutResponse Trap
  then unfold
```

4.2.1 Processing services Let us now describe the SNMP services. The service *Get* is used to fetch the value of a specific object. This service is defined in [2, section 4.2.2] as:

> *Upon receipt of a GetRequest-PDU, the receiving SNMPv2 entity processes each variable binding in the variable-binding list to produce a Response-PDU. All fields of the Response-PDU have the same values as the corresponding fields of the received request except as indicated below. Each variable binding is processed as follows:*
>
> 1. *If the variable binding's name exactly matches the name of a variable accessible by this request, then the variable binding's value field is set to the value of the named variable.*
> 2. *Otherwise, if the variable binding's name does not have an OBJECT IDENTIFIER prefix which exactly matches the OBJECT IDENTIFIER prefix of any (potential) variable accessible by this request, then its value field is set to 'noSuchObject'.*
> 3. *Otherwise, the variable binding's value field is set to 'noSuchInstance'.*
>
> *If the processing of any variable binding fails for a reason other than listed above, then the Response-PDU is re-formatted with the same values in its request-id and variable-bindings fields as the received GetRequest-PDU, with the value of its error-status field set to 'genErr', and the value of its error-index field is set to the index of the failed variable binding.*
>
> *Otherwise, the value of the Response-PDU's error-status field is set to 'noError', and the value of its error-index field is zero.*

The formal specification of each service is given by the "process _" actions. There is one action definition for each possible service of the SNMPv3 entity acting as an agent. For example, the description of the SNMP *Get* operation can be stated:

- process _ :: tag → action

```
(5)  process Get =
         |setError(noError, 0)
         and
         ||give the VarBindList bound to "variable-bindings"
         |then
         ||proceedList consultVarBind
        trap
         |setError(genErr, the given integer)
         and
         |give the VarBindList bound to "variable-bindings"
```

The auxiliary action "proceedList" takes an action A as parameter, and performs the action A with each of the elements of a given list as functional information. The results of the application of A to each element of the list are assembled in a list, which is the result of "proceedList".

The action combinator "A_1 trap A_2" behaves like the exception handling mechanisms in programming languages: The action A_1 is performed; in the case of an exceptional termination of A_1, the action A_2 is performed (otherwise, A_2 is ignored).

The action "consultVarBind" corresponds to the numbered items of the informal specification. It fetches the value stored in a single MIB object.

Let us give another example of the formal description of operations: $GetBulk$ is one of the most complex operations of the protocol. It is defined in [2, section 4.2.3].

The formal definition of the operation $GetBulk$, using Action Semantics, can be stated as:

```
(6)  process GetBulk =
         setError(noError, 0)
         and
             give min(the integer bound to "non-repeaters",
                      count items the VarBindList bound to "variable-bindings")
             and give the VarBindList bound to "variable-bindings"
             then
                 break(the given list#2, the given integer#1) then give the given list#1
                 then proceedList consultNextVarBind
                 and then
                     give the integer bound to "max-repetitions"
                     and
                     break(the given list#2, the given integer#1) then give the given list#2
                     then proceedListRepetition consultNextVarBind
             then give concatenation(the given list#1, the given list#2)
         trap
             setError(genErr, the given integer)
             and give the VarBindList bound to "variable-bindings"
```

Some auxiliary actions were used above: the action "break" takes a list L and an integer N as arguments, returning a pair of lists, such that:

- the concatenation of the two resulting lists is the list L.
- The number of elements of the first resulting list is N.

The auxiliary action "proceedListRepetition" is similar to "proceedList", with the only difference that the action A will be performed a given number of times over each element of a list.

The action "consultNextVarBind" is similar to "consultVarBind", but fetches the object which is next to the received object name in the MIB.

Its informal specifications is defined in [2, section 4.2.3] as:

> *Upon receipt of a GetBulkRequest-PDU, the receiving SNMPv2 entity processes each variable binding in the variable-binding list to produce a Response-PDU with its request-id field having the same value as in the request. Processing begins by examining the values in the non-repeaters and max-repetitions fields. If the value in the non-repeaters field is less than zero, then the value of the field is set to zero. Similarly, if the value in the max-repetitions field is less than zero, then the value of the field is set to zero.*
> *The values of the non-repeaters and max-repetitions fields in the request specify the processing requested. One variable binding in the Response-PDU is requested for the first N variable bindings in the request and M variable bindings are requested for each of the R remaining variable bindings in the request. Consequently, the total number of requested variable bindings communicated by the request is given by N + (M * R), where N is the minimum of: a) the value of the non-repeaters field*

in the request, and b) the number of variable bindings in the request; M is the value of the max-repetitions field in the request; and R is the maximum of: a) number of variable bindings in the request - N, and b) zero.
The receiving SNMPv2 entity produces a Response-PDU with up to the total number of requested variable bindings communicated by the request. The request-id shall have the same value as the received GetBulkRequest-PDU.
If N is greater than zero, the first through the (N)-th variable bindings of the Response-PDU are each produced as follows:

1. *The variable is located which is in the lexicographically ordered list of the names of all variables which are accessible by this request and whose name is the first lexicographic successor of the variable binding's name in the incoming GetNextRequest-PDU. The corresponding variable binding's name and value fields in the Response-PDU are set to the name and value of the located variable.*
2. *If the requested variable binding's name does not lexicographically precede the name of any variable accessible by this request, i.e., there is no lexicographic successor, then the corresponding variable binding produced in the Response-PDU has its value field set to 'endOfMibView', and its name field set to the variable binding's name in the request.*

*If M and R are non-zero, the (N + 1)-th and subsequent variable bindings of the Response-PDU are each produced in a similar manner. For each iteration i, such that i is greater than zero and less than or equal to M, and for each repeated variable, r, such that r is greater than zero and less than or equal to R, the (N + ((i-1) * R) + r)-th variable binding of the Response-PDU is produced as follows:*

1. *The variable is located which is in the lexicographically ordered list of the names of all variables which are accessible by this request and whose name is the first lexicographic successor of the variable binding's name in the incoming GetNextRequest-PDU. The corresponding variable binding's name and value fields in the Response-PDU are set to the name and value of the located variable.*
2. *If the requested variable binding's name does not lexicographically precede the name of any variable accessible by this request, i.e., there is no lexicographic successor, then the corresponding variable binding produced in the Response-PDU has its value field set to 'endOfMibView', and its name field set to the variable binding's name in the request.*

*While the maximum number of variable bindings in the Response-PDU is bounded by N + (M * R), the response may be generated with a lesser number of variable bindings (possibly zero) for either of three reasons.*
(1) If the size of the message encapsulating the Response-PDU containing the requested number of variable bindings would be greater than either a local constraint or the maximum message size of the originator, then

the response is generated with a lesser number of variable bindings. This lesser number is the ordered set of variable bindings with some of the variable bindings at the end of the set removed, such that the size of the message encapsulating the Response-PDU is approximately equal to but no greater than either a local constraint or the maximum message size of the originator. Note that the number of variable bindings removed has no relationship to the values of N, M, or R.
*(2) The response may also be generated with a lesser number of variable bindings if for some value of iteration i, such that i is greater than zero and less than or equal to M, that all of the generated variable bindings have the value field set to the 'endOfMibView'. In this case, the variable bindings may be truncated after the (N + (i * R))-th variable binding.*
(3) In the event that the processing of a request with many repetitions requires a significantly greater amount of processing time than a normal request, then an agent may terminate the request with less than the full number of repetitions, providing at least one repetition is completed.
If the processing of any variable binding fails for a reason other than listed above, then the Response-PDU is re-formatted with the same values in its request-id and variable-bindings fields as the received GetBulk-Request-PDU, with the value of its error-status field set to 'genErr', and the value of its error-index field is set to the index of the variable binding in the original request which corresponds to the failed variable binding. Otherwise, the value of the Response-PDU's error-status field is set to 'noError', and the value of its error-index field to zero.

4.3 Command Generators

Managing entities generate commands to be sent to agents. The function of each command generator application depends on the purpose of the service to be offered. In this work, we describe the SNMPv3 operations, which are to be included in command generator applications.

The "snmpGet(_,_)" action describes the manager side of the *Get* protocol operation. It can be specified as follows:

- snmpGet(_, _) :: entity, list of ObjectName^{+} → action

(7) snmpGet(E:entity, N:list of ObjectName^{+}) =
    |generateVarBindList from N
    then
    |request Get [to E][containing (0, 0, the given VarBindList)]

This action generates a "VarBindList", in accordance to [2, section 4.2.7], and then requests the service. The action "request" is defined next:

- request _ [to _][containing (_)] :: tag, agent, (integer, integer, VarBindList) → action

(8) request Op:tag [to E:entity][containing (S:integer, I:integer, V:VarBindList)]
=
|generateRequestID
then
|send a message[to E]
|[containing a PDU of (the given integer,S,I,V) with tag Op]
then
|accept a message[from E][containing a PDU tagged with Response]
then
|give (the given integer#3, the given integer#4, the given VarBindList#5)

This action sends a message to the agent responsible for the service and waits for an answer. As required in the informal specification, the message includes a request ID. The "request" action returns a triple containing an *error-status*, an *error-index* and a *variable-bindings*.

The formal specification of the complete set of SMNP operations (including "snmpGet", "snmpGetBulk", "snmpSet", "snmpInform" and "snmpTrap") will be soon available.

4.4 Notification Originators

Notification originators are applications that monitor specific conditions of the managed devices. These applications send notification messages to specified managers. Notification originators primitives can be specified in the same way as the operations in section 4.3.

5 Conclusions

Although SNMPv3 is the Internet standard management architecture, the semantics of SNMP components is given informally in the IETF documents. In this paper we have used Action Semantics to formally describe the semantics of four SNMPv3 standard applications: the command generator, the command responder, the notification originator and the notification receiver. All management entities based on SNMPv3 use these standard applications to communicate with other entities using the protocol. Depending of the applications it uses, an entity may have different roles in the system.

Future work include fully defining the messages the entities exchange and the services offered by the engine, which the applications use to communicate. When these basic components are fully defined, it will be possible to produce formal specifications of the entities themselves. From those formal specifications, implementations of managers and agents can be generated automatically.

References

1. J.Case, M.Fedor, M.Schoffstall, and J.Davin. A Simple Network Management Protocol (SNMP). Request for Comments 1157, May 1990.
2. J.Case, K.McCloghrie, M.Rose, and S.Waldbusser. Protocol Operations for Version 2 of the SNMP. Request for Comments 1905, January 1996.
3. J.Case, K.McCloghrie, M.Rose, and S.Waldbusser. Structure of Management Information for Version 2 of the SNMP. Request for Comments 1902, January 1996.
4. D.Harrington, R.Presuhn, and B.Wijnen. An Architecture for Describing SNMP Management Frameworks. Request for Comments 2571, May 1999.
5. E.P.Duarte Jr. and M.A.Musicante. Formal specification of snmp mib's using action semantics: The routing proxy case study. In *Proc. of the Sixth IFIP/IEEE Int'l Symp. on Integrated Network Management*, pages 417–430, Boston, USA, May 1999. IEEE Publishing.
6. E.P.Duarte Jr., G.Mansfield, T.Nanya, and S.Noguchi. Improving the dependability of network management systems. *International Journal of Network Management*, 8:244–253, 1998.
7. D.C.Furlan, M.A.Musicante, and E.P.Duarte Jr. An Action Semantics Description of the SNMPv3 Dispatcher. *Prod. of SBLP2000, IV Brasilian Symposium on Programming Languages*, 186–199, 2000.
8. S.B.Lassen. Action semantics reasoning about functional programs. Technical report, University of Aarhus, Department of Computer Science, 1995. Available at URL: `http://www.brics.dk/~thales/docs/ldpl.dvi.gz`.
9. D.Levi, P.Meyer, and B.Stewart. SNMPv3 Applications. Request for Comments 2573, May 1999.
10. P.D.Mosses. *Action Semantics*. Cambridge University Press, Cambridge, UK, 1992.
11. P.D.Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
12. P.D.Mosses and M.A.Musicante. An action semantics for ML concurrency primitives. Number 873 in Lecture Notes in Computer Science, Barcelona, Spain, October 1994. FME, Springer-Verlag.
13. M.A.Musicante. The Sun RPC language semantics. In *Proceedings of PANEL'92, XVIII Latin-American Conference on Informatics*. Universidad de Las Palmas de Gran Canaria, 1992.
14. J.P.Nielsen and J.U.Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DCC International A/S, Lundtoftvej 1C, DK-2800 Lyngby, Denmark, 1994.
15. M.T.Rose. *The Simple Book: an Introduction to Internet Management*. Prentice Hall, second edition, 1994.
16. D.A.Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1985.
17. D.A.Watt. *Programming Language Syntax and Semantics*. Prentice Hall, UK, 1991.

Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic (Abstract)

Christiano de O. Braga[1,2], E. Hermann Haeusler[2], José Meseguer[1], and Peter D. Mosses[3]

[1] Computer Science Laboratory, SRI International
[2] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro
[3] BRICS & Department of Computer Science, University of Aarhus

Abstract. Action semantics (AS) is a framework for specifying the semantics of programming languages, in a very modular and readable way. Recently, the operational semantics of action notation (action semantics's specification language) has been rewritten using Modular SOS (MSOS), a new modular approach for specifying operational semantics. The new modular specification of action notation facilitates the creation of extensions to action semantics, to deal with new concepts, such as components. The Maude Action Tool uses the reflective capabilities of rewriting logic, implemented on the Maude system, to create an executable environment for action semantics and its potential extensions.

This is achieved by a mapping between the MSOS and rewriting logic formalisms which, when applied to the MSOS semantics of each facet of action notation, yields a corresponding rewrite theory. Such rewrite theories are executed on *action programs*, that is, on the action notation translation of a given program P in a language L, according to L's action semantics.

We refer to [1] for a short presentation of the frameworks used in the mapping and the description of a prototype implemented in the Maude system. An extended version of [1], with a formal explanation on the mapping is available as a technical report at PUC-Rio University and can be requested via email to Christiano Braga, `cbraga@inf.puc-rio.br`.

References

1. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST 2000, Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology, Iowa City, Iowa, USA*, volume 1816 of *LNCS*, pages 407–421. Springer-Verlag, 2000.

A Modular Implementation of Action Notation

L. M. de Moura, C. J. P. de Lucena and E. H. Haeusler

Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rua Marquês de São Vicente, 225, 22543-900 Rio de Janeiro, Brazil
moura@les.inf.puc-rio.br

Abstract. This paper describes the implementation of a modular interpreter for the new modular semantics of action notation, the formal notation used in *action semantics*. Our interpreter supports nondeterministic features of action semantics, and allows the user to explore different execution traces. The implementation is based on a new logic programming language called PAN. A brief and pragmatic introduction to modular structured operational semantics is also provided.

1 Introduction

We present here an implementation of a modular interpreter for the new modular semantics of action notation, the formal notation used in *action semantics* [8]. Such implementation involves the use of a new logic programming language called PAN which we specially developed to implement language interpreters, analysis and verification tools. PAN has several features which are extremely useful for specifying the operational semantics of programming languages. An interpreter for action notation is particularly interesting, since it allows us to experiment in the early stages of the language design, which is very useful to consolidate our understanding of the language design.

Action Semantics [8] is a framework for the formal description of programming languages. Not only denotational semantics [16, 20], but also action semantics are compositional, *i.e.* the semantics of each phrase is determined by the semantics of its sub-phrases. The difference is that the semantics of phrases are *actions* instead of higher-order functions, which are commonly used in denotational semantics descriptions. Thus, action semantics might be regarded as denotational, where the denotations are actions. Action Notation, the notation used in action semantics descriptions, has an operational semantics, and nice algebraic properties [8]. The main concept in action semantics is the concept of action.

In structural operational semantics (SOS) [13] literature modularity issues have largely been neglected, and SOS descriptions of programming languages typically exhibit rather poor modularity. Thus, the original operational semantics of action notation was not modular. Therefore, we would not be able to reuse the existing semantic description when extending or changing the action notation. Recently, Mosses described how to obtain a high degree of modularity in SOS [9], and the SOS of Action Notation is currently being reformulated in the proposed modular style.

2 A Brief Introduction to MSOS

In this section we provide a brief introduction to the method of *Modular Structural Operational Semantics* (MSOS), which is a modular variation of SOS. SOS represents computations by means of deductive systems, transforming an abstract machine into a system of logical inferences. SOS definitions are given by inference rules consisting of conclusions that follow from a set of premises. Such rules describe possible transitions (*small steps*) of a computation. Each proof in the deductive system describes a state transition.

SOS involves abstract syntax, computed values, configurations and inference rules for (labelled) transitions. A SOS description specifies a *labelled transition system* $(\Gamma, T, A, \rightarrow)$, where Γ is a set of configurations, $T \subseteq \Gamma$ is a set of terminal configurations, A is a set of labels, and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is a transition relation. For configuration $\gamma, \gamma' \in \Gamma$ and labels $\alpha \in A$, the assertion that $(\gamma, \alpha, \gamma')$ is in the transition relation is written as $\gamma \xrightarrow{\alpha} \gamma'$.

On the other hand, for our objectives SOS descriptions can be seen as interpreters implemented in a logic programming language. The main predicate of these logical programs is the one which defines the transition relation.

However, most of the SOS descriptions lack modularity. Local modifications in language semantics usually imply global modifications in SOS descriptions. This also means that it is almost impossible to *reuse* SOS descriptions when defining a new language. To cope with this problem, we use an approach similar to the one defined in [9]. In this approach, the modularity is achieved by using encapsulation techniques. Figure 1 shows a small fragment of a modular SOS description for a simple imperative language. In this example, the configurations are restricted to abstract syntax trees, where nodes may be replaced by the values they computed, as in conventional SOS. The initial configurations are pure syntax, and terminal configurations are simply computed values. Value *completed* represents normal termination. All usual semantic components of configurations, such as stores, are incorporated into the labels on transitions. All labels contain a pair of stores reflecting the value of the store before and after the transition. The structure of the label is encapsulated, and should only be accessed by means of predicates. We can view the label as an object [1] and the predicates as the interface (methods) of the object.

All labels contain a partial composition predicate used to assert whether two transitions may be adjacent. In the example in Figure 1, this predicate is defined as:

$$compose(\alpha_1, \alpha_2, \alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha_1 = \langle store_1, store_2 \rangle \wedge \\ \alpha_2 = \langle store_2, store_3 \rangle \wedge \\ \alpha \ \ = \langle store_1, store_3 \rangle \end{cases}$$

This predicate asserts that label α is the composition of labels α_1 and α_2. Notice that in [9] such predicate is represented by the operation $_;_$, thus we may see $\alpha = \alpha_1; \alpha_2$ as a syntactic sugar for $compose(\alpha_1, \alpha_2, \alpha)$. Every label can always be composed on the left and on the right with a specific identity label. The

[1] Using the object-oriented terminology.

$$skip \xrightarrow{i} completed \qquad \frac{set\text{-}store(\alpha, Id, Val)}{Id := Val \xrightarrow{\alpha} completed}$$

$$\frac{Be \xrightarrow{\alpha} Be'}{if\ Be\ then\ S_1\ else\ S_2 \xrightarrow{\alpha} if\ Be'\ then\ S_1\ else\ S_2}$$

$$if\ true\ then\ S_1\ else\ S_2 \xrightarrow{i} S_1 \qquad if\ false\ then\ S_1\ else\ S_2 \xrightarrow{i} S_2$$

$$while\ E\ do\ S \xrightarrow{i} if\ E\ then\ S; while\ E\ do\ S\ else\ skip$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} S_1'; S_2} \qquad completed\ ; S_2 \xrightarrow{i} S_2$$

Fig. 1. Modular SOS example

notion of identity labels is defined by the predicate *id*. In the given example, this predicate is defined as:

$$id(\alpha) \stackrel{\text{def}}{=} (\alpha = \langle store, store \rangle)$$

We use $\gamma \xrightarrow{i} \gamma'$ as a syntax sugar for:

$$\frac{id(\alpha)}{\gamma \xrightarrow{\alpha} \gamma'}$$

As already mentioned, all labels should provide predicates to access their internal structure. In the given example, the label contains two predicates:

$$set\text{-}store(\alpha, Id, Val) \stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ S_2 = S_1[Id \leftarrow Val]$$
$$get\text{-}store(\alpha, Id, Val) \stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ Val = S_1(Id)$$

The predicate *set-store* "modifies" the value of a given variable in the store. The predicate *get-store* "accesses" the value of a given variable.

Using this approach, we do not need to completely modify a SOS description when we extend or modify the language semantics.

3 The *PAN* Programming Language

In our implementation, the operational semantics of a language must be specified in a programming language called *PAN*. PAN is a logic programming language specially developed to specify interpreters, program analyzers and verifiers. The PAN syntax and semantics have some similarities with *Prolog* [19]. The PAN compiler uses global analysis to produce code with a performance comparable to the one produced by imperative language compilers.

All PAN programs must be properly qualified by providing the kind, type and mixfix declarations. For instance, consider the following specification of the concatenation of lists in PAN syntax:

```
append nil X X.
append (X :: Xs) Ys (X :: Zs) :- append Xs Ys Zs.
```

This specification contains four variables `X`, `Xs`, `Ys` and `Zs`, one logical connective, namely `:-` for the converse of implication, and three non-logical connectives, namely `nil`, `::` and `append`, denoting the empty list, the list constructor, and the concatenation relation, respectively. Two of these connectives, namely `:-` and `::` are also used as infix symbols. This specification is only meaningful if mixfix declarations and type declarations are given for these connectives.

```
type nil : list A.
type _ :: _ : A -> list A -> list A {prec 8 r-assoc}.
type append : list A -> list A -> list A -> o.
```

PAN has a polymorphic type system similar to the type system of ML [7]. In type declarations, tokens with an initial upper case letter denote *type variables*. The type declaration for `nil` above asserts that for every type `A`, the symbol `nil` is of type `(list A)`. The type declaration for `_ :: _` asserts that given a term `H` of type `A`, and a term `T` of type `list A`, then the term `T :: H` is of type `list A`. If the target type of a type declaration is `o`, then the connective is a predicate symbol. In the above example, `append` is a predicate symbol that takes three arguments.

The underbars in connectives like `_ :: _` are place holders that indicate where terms of type `A` and `list A` should go, respectively. The precedence attribute `prec` n tells how tightly a symbol binds. The higher the number the tighter the symbol binds. The attributes `r-assoc` and `l-assoc` tell if a symbol is right or left associative. Precedence and associativity information is used to disambiguate the parsing. Notice that the above declarations have introduced two new constants not in the original specification, namely `o` and `list`. These type constants will need declarations, called *kind declarations*. Kind declarations are used to introduce *type constructors*. An example of kind declaration is:

```
kind list : type -> type.
```

Here, `list` takes a type and returns another type. For example, `list (list int)` is the type of a list of lists of integers.

PAN also supports higher orders predicates, i.e. predicates that have other predicates as arguments. For example:

```
type map : (A -> B -> o) -> list A -> list B -> o.
map P nil nil.
map P (X :: Xs) (Y :: Ys) :- (P X Y), map P Xs Ys.
```

The goal `(map F Xs Ys)` is provable, if `Xs` and `Ys` are lists of equal length and corresponding elements of these lists are related by the predicate `P`.

Prolog has various all-solutions predicates (e.g. findall/3), all of them having semantic problems. PAN has the builtin predicate called *solutions*. We avoid the variable scoping problems of most Prolog versions. Rather than taking a goal to execute and an aliased term holding the resulting value to collect, the predicate *solutions* take as input a single higher-order predicate. The declaration of the predicate *solutions* is:

```
type solutions _ _  : ((A0 -> o) -> (list A0) -> o)
```

PAN compiler is based on state of the art logic programming compilation technology [14, 18]. The compiler uses several global analyses (mode analyses, determinism analyses, switch detection, etc) to produce efficient code. It is out of scope of this article to explain each one of such analyses algorithms. The output of the PAN compiler is portable C code. Hence, the PAN compiler allows us to invoke C code from PAN code and vice-versa.

PAN does not have "impure" features like Prolog [2]. We avoid the efficiency problems by using an approach similar to the one found in pure functional languages [21]. Briefly, the user can decorate the code with uniqueness marks. Such marks tell the compiler that a given argument of a given predicate can only have one variable referencing it. As there is only one reference to a given argument, destructive updates can be safely performed. This safety condition is checked by the compiler, and if not satisfied produces a warning message. This feature is used to allow PAN code to invoke C code which performs destructive updates in a safe way. Figure 2 shows an example of interface between PAN and C code. The first three lines show how to instruct the compiler to add arbitrary C code into the produced code. The *cdecl* declarations tell the compiler that the respective predicates are implemented by using C code. The marks *in* and *out* are used to indicate which arguments are input or output. The uniqueness marks are represented by the symbol *unq*. For instance, the predicate *update-array* receives an index, a value and the array that will be updated. The resulting array is returned in the fourth argument.

The PAN features are extremely useful for specifying the operational semantics of a programming language. The mixfix notation allows us to specify the abstract syntax of a programming language and to describe the transition rules in a clear way. For instance, Figure 3 shows part of the abstract syntax definition of the simple imperative language.

Figure 4 shows part of the translation to PAN of the SOS description of Figure 1. The symbol `|-` is an alternative for `:-` that simplifies the description of SOS rules. It allows us to write a clause as `Body |- Head` instead of `Head :- Body`. The predicate `_ -- _ --> _` specifies the transition relation.

4 Implementing Action Notation

Action Notation [8] is a rich algebraic notation for expressing actions, which are used to represent the semantics of constructs of conventional programming

[2] The only non-logical feature of the PAN language is the *cut*.

```
C{{
#include "array.h"
}}C.
kind array : type -> type.

cdecl create-empty-array : unq out array A -> o {det}
C{{ // C code ... }}C.

cdecl update-array : in int -> in A -> unq in array A ->
                     unq out array A -> o {det}
C{{ // C code ... }}C.
```

Fig. 2. Interface with the C language

```
kind stmt : type.
kind expr : type.

type _ ; _ : stmt -> stmt -> stmt {prec 80 l-assoc}.
type skip : stmt.
type if _ then _ else _ end :
                 expr -> stmt -> stmt -> stmt {prec 65}.
type while _ do _ end : expr -> stmt -> stmt {prec 70}.
type _ := _ : id -> expr -> stmt {prec 100}.
```

Fig. 3. Abstract syntax definition

```
kind label : type.
kind config : type.
sub stmt : config.
type completed : config.
type _ -- _ --> _ : config -> label -> config -> o.

S1 -- L --> S1'
|-
S1 ; S2 -- L --> S1' ; S2.

id L
|-
while E do S end -- L --> if E then S ; (while E do S end) end.

(is-val E V), (set-store L ID V)
|-
ID := E -- L --> completed.
```

Fig. 4. Fragment of language semantics

languages. Such notation is verbose and suggestive, which improves readability of semantic descriptions.

The performance of an action directly represents information processing behavior and reflects the gradual, stepwise nature of computation: each step of an action performance may access and/or change the current information. Yielders occurring in actions may access, but not change, the current information.

A performance of an action either: completes, corresponding to normal termination; or escapes, corresponding to exceptional termination; or fails, corresponding to abandoning an alternative; or diverges.

Action notation consists of several independent parts called *facets*. AN contains the following facets:

Basic: is used to specify the flow of control in actions;
Functional: is used to specify the flow of the data;
Declarative: is used to specify the scopes of the bindings that are received and produced by actions;
Imperative: is used to specify the allocation of storage for values of variables;
Reflective: is used to specify procedural abstraction and application;
Communicative: is used to specify message passing.

In our implementation, each facet is coded in an independent PAN module. Each module requires some features from the label data structure, *i.e.* the label definition module should implement some predicates which provide the desired services. For example, the basic facet requires the predicates: *set-commitment*, *get-commitment*, *set-unfolding*, and *get-unfolding*. Notice that in

the formal description of action semantics [10], each label provides the operations $set(\alpha, data, d)$ and $get(\alpha, data)$. In this way, the predicate call

$$set\text{-}commitment(\alpha, committed, \alpha')$$

used in our description is equivalent to

$$\alpha' = set(\alpha, commitment, committed)$$

The PAN module *action-support* (Figure 5) provides the basic definitions that are used in our implementation. Notice that the predicate `_ -- _ -->+ _` is the transitive closure of the predicate `_ -- _ --> _`.

```
module action-support.

kind action : type.
kind yielder : type.
kind label : type.

// basic label "methods"
type id : label -> o.
type compose : label -> label -> label -> o.

type _ -- _ --> _ : action -> label -> action -> o.
// definition of the transitive closure of _ -- _ --> _
type _ -- _ -->+ _ : action -> label -> action -> o.
type _ -- _ --> _ : yielder -> label -> yielder -> o.

A1 -- L --> A2
|-
A1 -- L -->+ A2.

A1 -- L1 --> A2,  A2 -- L2 -->+ A3, (compose L1 L2 L)
|-
A1 -- L -->+ A3.
```

Fig. 5. Fragment of the module *action-support*

The implementation of such modules is almost identical to the formal description of the facets [10]. For example, Figure 6 contains a fragment of the formal definition of the basic facet. Such fragment was extracted from [10]. As stated above, it is quite straightforward to code a SOS description in PAN. Figure 7 contains the definition of the abstract syntax tree in PAN. Figure 8 contains additional definitions. Note that we defined a new predicate called *is-terminated* which specifies if an action has been terminated or not. Figure 9 contains the

implementation of the module. It is clear that such implementation is almost identical to the formal definition described in Figure 6. The implementation of the other facets is also straightforward [3].

---

type *Terminated* ::= *completed* | *escaped* | *failed*
type *Action* ::= _ **or** _(*Action*; *Action*) | **fail** | **commit** |
_ **and** _(*Action*; *Action*) | **complete** |
indivisibly _(*Action*) | _ **and then** _(*Action*;*Action*) |
unfolding _(*Action*) | **unfold**
...
sort *Terminated* | _ **@** _(*Action*; *Action*)

$$\frac{A_1 \xrightarrow{\alpha} A_1'}{A_1 \textbf{ or } A_2 \xrightarrow{\alpha} A_1' \textbf{ or } A_2} \qquad \frac{A_2 \xrightarrow{\alpha} A_2'}{A_1 \textbf{ or } A_2 \xrightarrow{\alpha} A_1 \textbf{ or } A_2'}$$

$$\textit{completed} \textbf{ or } A_2 \xrightarrow{i} \textit{completed} \qquad A_1 \textbf{ or } \textit{completed} \xrightarrow{i} \textit{completed}$$

...

$$\frac{A_1 \xrightarrow{\alpha} A_1' \quad \textit{get}(\alpha, \textit{commitment}) = \textit{committed}}{A_1 \textbf{ or } A_2 \xrightarrow{\alpha} A_1'}$$

$$\frac{A_2 \xrightarrow{\alpha} A_2' \quad \textit{get}(\alpha, \textit{commitment}) = \textit{committed}}{A_1 \textbf{ or } A_2 \xrightarrow{\alpha} A_2'}$$

...

$$\frac{\textbf{var } t : \textit{Terminated} \qquad A \xrightarrow{\alpha}{}^{+} t}{\textbf{indivisibly } A \xrightarrow{\alpha} t}$$

...

$$\textbf{unfolding } A \xrightarrow{i} A \textbf{ @ } A \qquad t \textbf{ @ } A_0 \xrightarrow{i} t$$

$$\frac{\alpha' = \textit{set}(\alpha, \textit{unfolding}, A_0) \quad A \xrightarrow{\alpha'} A'}{A \textbf{ @ } A_0 \xrightarrow{\alpha} A' \textbf{ @ } A_0}$$

$$\frac{\textit{get}(i, \textit{unfolding}) = A_0}{\textbf{unfold} \xrightarrow{i} A_0}$$

...

Fig. 6. Fragment of the Basic Facet

The implementation of the *label* module was also quite simple. Basically, it contains the context (e.g. *transients* and *bindings*), mutable (e.g. *storage*) and emitted (e.g. *commitment*) information. Transients, bindings and storage are

```
// ----------- Abstract Syntax Tree ---------------
type _ or _ : action -> action -> action {prec 60 l-assoc}.
type _ and _ : action -> action -> action {prec 70 l-assoc}.
type indivisibly _ : action -> action {prec 50 l-assoc}.
type unfolding _ : action -> action {prec 50}.
type unfold : action.
type diverge : action.

type fail : action.
type complete : action.
type escape : action.
type commit : action.
```

Fig. 7. Fragment of the abstract syntax tree definition coded in PAN

```
// values added to the AST
type _ @ _ : action -> action -> action {prec 100 l-assoc}.
type completed : action.
type failed : action.
type escaped : action.
type nothing : yielder.

// auxiliar "method"
type is-terminated : action -> o.
is-terminated completed.
is-terminated failed.
is-terminated escaped.
```

Fig. 8. Additional definitions

```
A1 -- L --> A1'
|-
A1 or A2 -- L --> A1' or A2.

A2 -- L --> A2'
|-
A1 or A2 -- L --> A1 or A2'.

A1 -- L --> A1', (get-commitment L committed)
|-
A1 or A2 -- L --> A1'.

A2 -- L --> A2', (get-commitment L committed)
|-
A1 or A2 -- L --> A2'.

id L, (set-commitment L committed L')
|-
commit -- L' --> completed.

A -- L -->+ T, (is-terminated T)
|-
indivisibly A -- L --> T.

id L
|-
unfolding A -- L --> A @ A.

id L, is-terminated T
|-
T @ A -- L --> T.

(set-unfolding L A0 L'), A -- L' --> A'
|-
A @ A0 -- L --> A' @ A0.

(get-unfolding L A0)
|-
unfold -- L --> A0.
```

Fig. 9. Fragment of the Basic Facet coded in PAN

implemented by using *mappings*. The performance of our interpreter is improved by using the C language to implement the mapping data structure.

Notice that was not necessary to implement a parser for action notation, since the abstract syntax definition was sufficient due to the mixfix capabilities of the PAN language. For example, if we desire to obtain the result of performing action

```
| give 2
then
| give the successor of (the given integer)
```

we should simply perform the following PAN query

```
(give 2)
then
(give the successor of (the given integer)) -- L -->+ T,
is-terminated T.
```

The label `L` and the final configuration contains the outcome of the computation, *i.e. transients*, *bindings* and *storage*. The user may use the predicate *display-results* to print such mappings on the screen. It is important to remember that PAN does not have side effects, therefore we use an approach similar to the one used in the Haskell [5] language to implement IO. In this way, the signature of predicate *display-results* is:

```
type display-results: unq in io -> unq out io ->
                      in label -> in config -> o.
```

The first two parameters represents the state of the "world" (IO devices) before and after executing the predicate *display-results*. The uniqueness marks allows the predicate to perform destructives updates in the "world", *i.e.* to print on the screen.

4.1 Handling the nondeterminism

It is extremely simple to describe nondeterministic and concurrent languages by using SOS, differently from denotational semantics. Action Notation provides several nondeterministic combinators such as `_or_` and `_and_`. Therefore, it is important to define how the nondeterminism will be handled by our interpreter.

We say a language is nondeterministic, when a given state of a given program written in such language has more than one successor state [3]. In other words, a given program has more than an execution trace. We define *trace* as a finite or infinite sequence of states $(s_0, s_1, \ldots, s_i, \ldots)$, where there is a transition from s_i to s_{i+1}.

Thus, one interpreter (simulator) for such kind of language must clearly state how the successor state is chosen. A naive interpreter always selects a specific

[3] We say s' is a successor of s if there is a transition from s to s'

trace, and it is not capable of reproducing all possible behaviors of a given nondeterministic program.

Our implementation provides infrastructure to define different "flavors" of interpretation starting from the operational semantics. Our framework supports the following "flavors" of interpretation:

Standard: the successor state is chosen deterministically. The choice is based on the order in which the rules are described.

Random Selection: the successor state is chosen randomly. For deterministic languages this "flavor" behaves as the standard interpreter.

Guided: the user chooses the successor state.

Oracle: the user provides a function that chooses the successor state. Notice that such function may request user interaction in specific points of the simulation.

Abstractly, all "flavors" described above can be seen as instances of the algorithm described in Figure 10. The only difference is the way the function *select* is implemented. It is important to notice that the all possible successors of a state is obtained by using the builtin predicate *solutions*.

If desired, the user can also select interpretation algorithms that store the selected trace during the simulation, saving the *history* of the computation. We also provide support for common debugging operations like setting breakpoints and inspecting the value of program variables. Obviously, none of such features are described in Figure 10.

```
interpret (State s0)
{
  State s = s0;
  while (successors(s) ≠ ∅)
        s = select(successors(s));
}
```

Fig. 10. The "abstract" interpretation algorithm

4.2 Interpreting Programs

Although our interpreter may interpret arbitrary actions, it is much more useful to interpret actions which are denotations of programs. We may use two different approaches to handle such issue. The first approach uses the *actioneer generator* [11]. The actioneer generator is a program that given a semantic description of a language $\mathcal{L}$, it produces a program $\mathcal{P}$ which transforms abstract syntax trees of programs coded in $\mathcal{L}$ into their denotations (abstract syntax tree *ast* coded in

action notation). In order to interpret *ast*, we must first convert it into a format of the PAN language. Fortunately, such process is straightforward.

The other approach is similar to the one described in [17]. It is based on the fact that logic programming languages are extremely useful for applying programming transformations [19]. The following semantic equations:

execute $[\![C_1 \ ; \ C_2]\!]$ = execute C_1 and then execute C_2.
elaborate $[\![$ `var` I : $T]\!]$ =
    |allocate a cell
    then
    |bind I to the given cell

are coded in PAN as:

```
type execute [[ _ ]]= _ : Command -> Action -> o.
type elaborate [[ _ ]]= _ : Declaration -> Action -> o.

execute [[ C1 ; C2 ]]= ExecuteC1 and then ExecuteC2 :-
                  execute [[ C1 ]]= ExecuteC1,
                  execute [[ C2 ]]= ExecuteC2.

elaborate [[ var I : T ]]= (allocate a cell)
                           then
                           (bind I to the given cell).
```

By using such approaches, it is possible to interpret the abstract syntax of programs coded in a language $\mathcal{L}$. A parser for language $\mathcal{L}$ may be implemented by using tools such as YACC, allowing us to implement a complete interpreter for language $\mathcal{L}$. The interpreter coded in PAN may call the generated parser coded in C.

An interpreter should provide some kind of traceability, i.e. an association between the abstract syntax tree nodes and the original source code. Such feature is important if we desire to interact with the interpreter, and to perform actions such as setting breakpoints and selecting execution traces. Thus, our implementation uses mappings to associate abstract syntax tree nodes with source code information such as line and column numbers.

5 Future Work (Partial Evaluation)

We are currently developing an automatic partial evaluator for the PAN language based on the ideas used to implement *Mixtus* [4] [15]. The absence of "impure" features in the PAN language avoids several problems found in the implementation of Mixtus.

Partial evaluation is a program transformation which specializes a program to a particular context reducing its execution time and, in some cases, its size [6]. A

[4] Mixtus is an automatic partial evaluator for Prolog.

specialization context is defined by assigning values to some subset of a program's inputs.

Futamura [4] showed that a partial evaluator applied to an interpreter (for language X, written in a language Y) given a program will yield a new program (in language Y). This is called the first Futamura projection, and is probably the most useful and widely used aspect of partial evaluation, as the overhead of the interpreter might be eliminated.

More specifically, consider our interpreter $\mathcal{I}$ for action notation. The inputs of our interpreter are an action a and an initial state s_0 (*transients*, *bindings*, and *storage*). Now, using partial evaluation, we obtain $\mathcal{I}_a$ which is a specialization of $\mathcal{I}$ with respect to the action a. Notice that, $\mathcal{I}_a$ is a PAN program, and our compiler should be used to convert it in an equivalent C program.

Such process may be used to produce compiled code starting from action semantics descriptions. A similar approach using Scheme is described in [1].

6 Conclusion

We described a modular interpreter for action notation which was implemented by using a new logic programming language called PAN. Such interpreter may be used to obtain action semantics executable specifications.

The interpreter has several features to handle nondeterministic actions, and it allows the user to explore different execution traces. Such feature is interesting when handling nondeterministic or concurrent programming language. As far as we know, previous interpreters and compilers [11, 1, 12, 2] for action notation are only able to handle deterministic actions, *i.e.* the nondeterministic action primitives and combinators are handled (approximated) in a deterministic way.

Due to the modular implementation, new facets may be simply introduced in our implementation allowing us to extend and modify action semantics. For example, a new facet seems necessary to provide elegant semantic descriptions of complex concurrent languages.

References

1. A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.
2. D. F. Brown, H. Moura, and D. A. Watt. Actress: An action semantics directed compiler generator. In *Proc. of CC'92, 4th International Conference on Compiler Construction*, volume 641 of *LNCS*, pages 95–109. Springer-Verlag, 1992.
3. L. M. de Moura. Um framework para análise e verificação de programas. Ph.D. Thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 2000.
4. Y. Futamura. Partial evaluation of computation process – An approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
5. P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: a nonstrict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, Mar. 1992.

6. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
7. R. Milner. Theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17(3):348–375, 1978.
8. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
9. P. D. Mosses. Foundations of modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, LNCS, 1999.
10. P. D. Mosses. A modular SOS for Action Notation. Technical Report BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
11. H. Moura. An implementation of action semantics. Draft, 1993.
12. J. Palsberg. A provably correct compiler generator. In *Proc. of ESOP'92 European Symposium on Programming*, volume 582 of *LNCS*, pages 418–434. Springer-Verlag, Feb. 1992.
13. G. D. Plotkin. A Structural approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, Sept. 1981.
14. P. L. V. Roy and A. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, Jan. 1992.
15. D. Sahlin. An automatic partial evaluator for full Prolog. Ph.D. thesis TRITA-TCS-9101, Kungliga Tekniska Hgskolan, Stockholm, Sweden, 1991.
16. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.
17. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.
18. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
19. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.
20. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
21. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, pages 547–566, Sea of Galilee, Israel, Apr. 1990. North Holland. IFIP TC 2 Working Conference.

Recent BRICS Notes Series Publications

NS-00-6 **Peter D. Mosses and Hermano Perrelli de Moura, editors.** ***Proceedings of the Third International Workshop on Action Semantics, AS 2000,*** **(Recife, Brazil, May 15–16, 2000), August 2000. viii+148 pp.**

NS-00-5 **Claus Brabrand.** ***<bigwig> Version 1.3 — Tutorial*****. September 2000.**

NS-00-4 **Claus Brabrand.** ***<bigwig> Version 1.3 — Reference Manual*****. September 2000. ii+56 pp.**

NS-00-3 **Patrick Cousot, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raussen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '00,*** **(State College, USA, August 21, 2000), August 2000. vi+116 pp.**

NS-00-2 **Luca Aceto and Björn Victor, editors.** ***Preliminary Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00,*** **(State College, USA, August 21, 2000), August 2000. vi+130 pp.**

NS-00-1 **Bernd Gärtner.** ***Randomization and Abstraction — Useful Tools for Optimization*****. February 2000. 106 pp.**

NS-99-3 **Peter D. Mosses and David A. Watt, editors.** ***Proceedings of the Second International Workshop on Action Semantics, AS '99,*** **(Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.**

NS-99-2 **Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors.** ***Proceedings of the Workshop on Semantics of Objects As Processes, SOAP '99,*** **(Lisbon, Portugal, June 15, 1999), May 1999. iv+64 pp.**

NS-99-1 **Olivier Danvy, editor.** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '99,*** **(San Antonio, Texas, USA, January 22–23, 1999), January 1999.**

NS-98-8 **Olivier Danvy and Peter Dybjer, editors.** ***Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98 Proceedings,*** **(Gothenburg, Sweden, May 8–9, 1998), December 1998.**