BRICS RS-04-36 van den Brand et al.: An Action Environment

Basic Research in Computer Science

# An Action Environment

Mark van den Brand
Jørgen Iversen
Peter D. Mosses

BRICS Report Series RS-04-36
ISSN 0909-0878 December 2004

**Copyright © 2004, Mark van den Brand & Jørgen Iversen & Peter D. Mosses.**
**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 Report 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** RS/04/36/

# An Action Environment

Mark van den Brand

*Department of Software Engineering, CWI*
*Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands*
*and*
*Instituut voor Informatica, Hogeschool van Amsterdam*
*Weesperzijde 190, NL-1097 DZ Amsterdam, The Netherlands*

Jørgen Iversen, Peter D. Mosses

*BRICS & Department of Computer Science* [1]
*University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark*

**Abstract**

Some basic programming constructs (e.g., conditional statements) are found in many different programming languages, and can often be included without change when a new language is designed. When writing a semantic description of a language, however, it is usually not possible to reuse parts of previous descriptions without change.

This paper introduces a new formalism, ASDF, which has been designed specifically for giving reusable action semantic descriptions of individual language constructs. An initial case study in the use of ASDF has already provided reusable descriptions of all the basic constructs underlying Core ML.

The paper also describes the Action Environment, a new environment supporting use and validation of ASDF descriptions. The Action Environment has been implemented on top of the ASF+SDF Meta-Environment, exploiting recent advances in techniques for integration of different formalisms, and inheriting all the main features of the Meta-Environment.

*Key words:* ASF+SDF, ASDF, action semantics, modularity, reuse, language environment

*Email addresses:* `Mark.van.den.Brand@cwi.nl` (Mark van den Brand), `j.iversen@brics.dk` (Jørgen Iversen), `pdmosses@brics.dk` (Peter D. Mosses).

[1] Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

## 1 Introduction

Action Semantics [19] is a practical framework for describing the dynamic semantics of programming languages. The part of an action semantic description (ASD) concerned with any particular construct is independent of what other constructs are included in the described language, so ASDs enjoy a high degree of inherent modularity, and can easily be extended or modified. It is also possible to reuse parts of the ASD of one language in the ASD of another, without change. With the conventional modular structure of an ASD, however, it is usually not possible to reuse entire modules, so one has to copy and paste the required parts.

Doh and Mosses [12] proposed a flatter modular structure for ASDs, with the description of each construct being a separate module. This new structure allows a complete language to be described simply by listing the names of the modules for the included constructs, and fully supports explicit reuse of parts of semantic descriptions. Doh and Mosses formulated their modules in ASF+SDF [11], and used the ASF+SDF Meta-Environment [7] for checking them.

The approach of Doh and Mosses was feasible, but the direct use of ASF+SDF carried a considerable notational overhead. In this paper, we introduce a new action semantic description formalism, ASDF, which has been designed specifically for giving reusable descriptions of individual language constructs. We also report on the Action Environment, a new environment supporting use and validation of ASDF descriptions. The Action Environment has been implemented on top of the ASF+SDF Meta-Environment, exploiting recent advances in techniques for integration of different formalisms [6], and inheriting all the main features of the Meta-Environment. This was feasible due to the open architecture of the Meta-Environment. The Meta-Environment has a component-based architecture which allows an easy connection of new components in a fairly easy manner. In order to transform the ASF+SDF Meta-Environment into the Action Environment, a number of new components had to be defined, and plugged into the Meta-Environment. The most important ones were the components that took care of translating ASDF into ASF+SDF.

The present paper is an extended version of [5]. We have added documentation of how the Action Environment has been improved within the last year, which include support for both ASF+SDF and ASDF, and that an ASDF type checker and an action interpreter have been connected to the environment.

**Overview:** Section 2 recalls ASF+SDF and the Meta-Environment. Section 3 gives a brief outline of Action Semantics, focusing on modularity. Section 4 introduces ASDF and the Action Environment. Section 5 recalls the archi-

tecture of the Meta-Environment, and explains the novel techniques used to integrate ASDF. Section 6 mentions some related work. Section 7 concludes.

## 2 ASF+SDF

ASF+SDF is a general-purpose, executable, algebraic specification language. Its main application area has hitherto been in the modular definition of the syntax and the static semantics of (programming) languages, but it has also been used for the modular definition of (dynamic) action semantics of languages (see Section 3) and for defining translations between languages.

As the name indicates, the ASF+SDF formalism is a combination of two previous formalisms: ASF, the Algebraic Specification Formalism [2, 11], and SDF, the Syntax Definition Formalism [13]. SDF is used to define the concrete syntax of a language, whereas ASF is used to define conditional rewrite rules; the combination ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of so-called 'mixfix notation' in algebraic specifications. ASF+SDF allows specifications to be divided into named modules, facilitating reuse and sharing (as in SDF).

In the rest of this section, both SDF and ASF will be discussed, as well as the interactive programming environment that supports the use of ASF+SDF: the ASF+SDF Meta-Environment [7].

### *2.1 Syntax Definition Formalism*

The Syntax Definition Formalism SDF is a declarative formalism used to define concrete syntax of languages: not only programming languages, e.g., Java and COBOL, but also specification languages, e.g., CASL, Elan, and Action Semantics. In contrast to (E)BNF-like formalisms, SDF allows a modular definition of grammars. Furthermore, SDF does not impose a specific class of grammars, like LL(k), LR(k), etc., but allows arbitrary, cycle-free, context-free grammars — the grammars may even be ambiguous. The choice of the class of arbitrary context-free grammars enables the modular definition of grammars, because only this class is closed under union. Although the full power of arbitrary context-free grammars is hardly necessary when defining the syntax of a programming language (except for languages like COBOL, PL/I, etc.), modularity is essential for reuse of specific language constructs in various language definitions.

An SDF definition consists of a collection of modules where modules may im-

port other modules. The import mechanism offers primitive parameterisation and symbol-renaming facilities. This is demonstrated in Figure 1: The formal parameter `X` of the module "`containers/List`" is instantiated with the actual parameter `Integer`. The imported modules are automatically exported; the syntax defined in the module can either be exported or hidden.

```
module ListOfIntegers
imports basic/Integers containers/List[Integer]
  ...

module containers/List[X]
imports basic/Booleans basic/Integers
  ...
```

Fig. 1. A small SDF definition demonstrating the parameterisation mechanism

Figure 2 demonstrates the basic SDF features for defining lexical syntax, context-free syntax, associativities, and priorities.

```
module basic/Integers
imports basic/Booleans

exports
  sorts NatCon Integer
  lexical syntax
    [0-9]+ -> NatCon
  context-free syntax
    NatCon                 -> Integer
    Integer "+" Integer    -> Integer {left}
    Integer "-" Integer    -> Integer {left}
    Integer "*" Integer    -> Integer {left}
    "(" Integer ")"        -> Integer {bracket}

  context-free priorities
    Integer "*" Integer -> Integer >
    {left:  Integer "+" Integer -> Integer
            Integer "-" Integer -> Integer}

  lexical restrictions
    NatCon -/- [0-9]

hiddens
  ...
  variables
    "Int"[0-9]* -> Integer
```

Fig. 2. An SDF module of the Integers

### 2.2 Algebraic Specification Formalism

The Algebraic Specification Formalism ASF provides conditional equations, where also negative conditions are allowed, see Figure 3 for a number of ASF equations for the Integers. The concrete syntax defined in the corresponding SDF module and in the transitive closure of the imported modules (only the exported sections, of course) can be used when writing the conditional equations of an ASF module.

```
equations

  [] 0 + Int = Int
  [] Int + 0 = Int
  [] 1 + 1 = 2
  [] 1 + 2 = 3
  ...
  [] Int * 0 = 0
  [] Int * 1 = Int
  [] gt(Int2, 1) = true
     ====>
     Int1 * Int2 = Int1 + Int1 * (Int2 - 1)
   ...
```

Fig. 3. Some ASF equations for the Integers

### 2.3 The ASF+SDF Meta-Environment

The development of ASF+SDF specifications is supported by an interactive integrated programming environment, the ASF+SDF Meta-Environment [7]. This programming environment provides syntax directed editing facilities for both the SDF and ASF parts of modules as well as for terms, well-formedness checking of modules, interactive debugging of ASF equations, and visualisation facilities of the import graph and parse trees. The environment offers all kinds of refactoring operations at the specification level: renaming of modules, copying of modules, etc. Furthermore, a library of predefined primitive data structures, e.g., Booleans, Integers, Strings, Lists, Sets, etc., is available. The library contains also a growing collection of grammars of programming and specification languages, e.g., Java, C, CASL, SDF itself, etc.

The user interface of the ASF+SDF Meta-Environment is shown in Figure 4. Modules defining the concrete syntax of Pico (a toy language) have been opened. In the left part we see a tree-structured view of the modules, whereas the right pane shows the graph with import relations of the modules.

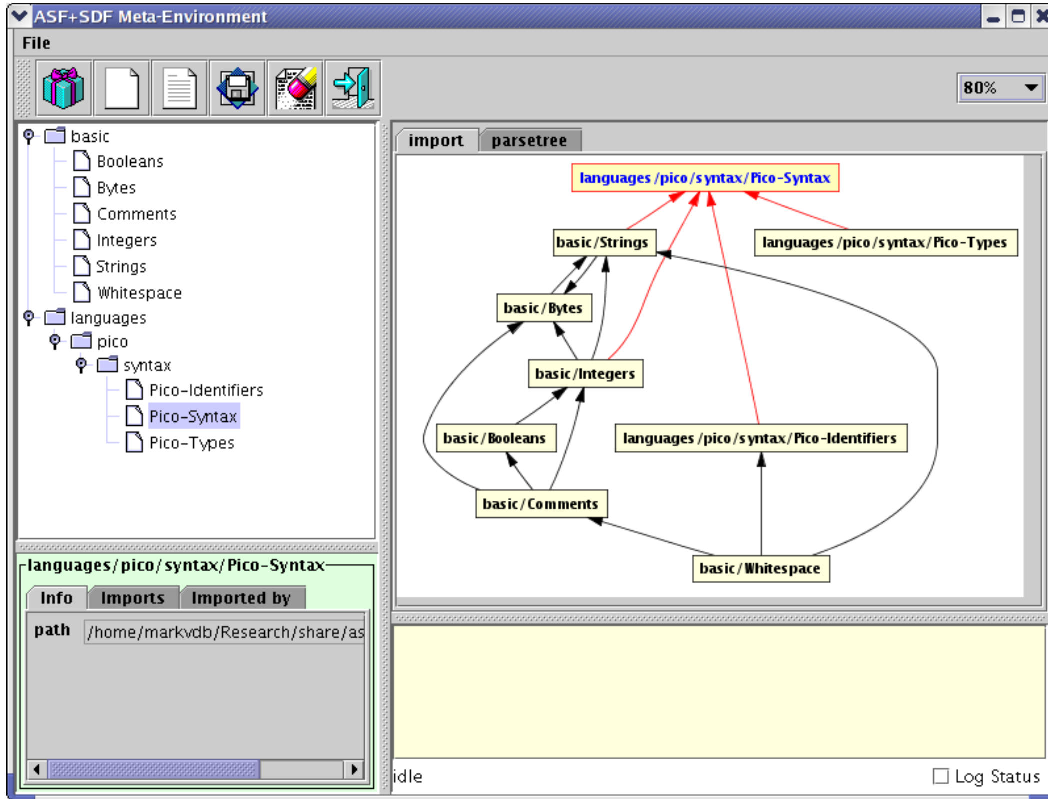



Fig. 4. GUI of the ASF+SDF Meta-Environment.

## 3 Action Semantics

The main aim of Action Semantics [19] is that descriptions of programming languages should be as easy as possible to work with. Action semantic descriptions (ASDs) scale up smoothly from small idealised languages to full languages [8, 22], and they have a high degree of comprehensibility (regarding not only perspicuity of notation, but also underlying concepts). They also have inherently good modularity, and can be extended or modified without reformulation of those parts of the description concerned with the unchanged constructs.

Action Semantics (AS) is a hybrid of Denotational Semantics and Operational Semantics, and combines the best features of both approaches. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations, which model their behaviour. The difference is that here, denotations are *actions*, and expressed in Action Notation (AN), which is itself defined operationally (originally [19, App. C] using Structural Operational Semantics, later [20] in a more modular style).

The inherent modularity of ASDs comes from the design of AN, not from their

explicit division into named modules. For instance, applications of action combinators remain valid (and meaningful) when the actions that they combine are enriched with new facets of behaviour; and similarly regarding the data processed by actions. The original version of AN [19, App. B] was rather large, but the revised version, AN-2 [16], is much more economical, and the size of the AN-2 kernel notation is comparable to that of the notation used in the monadic style of denotational semantics (e.g., as used in [17]).

Although the division of an ASD into named modules is not essential for extensibility and modifiability, the overall modular structure is of crucial significance for *reusability.* The original structure of ASDs was hierarchical, being a refinement of the usual division of semantic descriptions into sections dealing with abstract syntax, auxiliary semantic entities, and definitions of semantic functions. The abstract syntax module was divided into submodules, one for each sort of construct (expressions, statements, etc.), and similarly for the semantic functions; the submodules for the auxiliary entities were similarly focused on particular sorts of data. The implementation of a previous environment for AS based on the ASF+SDF Meta-Environment, the ASD Tools [10], relied on this structure to distinguish between the different kinds of submodules.

Doh and Mosses [12] realized that this conventional modular structure was a major impediment to explicit reuse of parts of ASDs. For example, suppose that an AS for Standard ML has already been given [22], and we are writing an AS for Java [8]. We cannot import the entire module for expressions from the ASD of ML for reuse in the ASD of Java, since this would include ML constructs not found in Java (e.g., anonymous function abstractions). On the other hand, if we were simply to copy and paste the individual semantic equations for the common constructs, this would leave no explicit indication of the fact that the two languages do have constructs in common, and readers of the two descriptions would have to compare the details of the semantic equations to discover exactly which the common constructs are.

Doh and Mosses proposed a radical change to the modular structure of ASDs, to support an incremental approach to semantics and allow explicit reuse of parts of ASDs. The main idea was to introduce a separate module for each *individual construct*, specifying both its syntax and the semantic equation defining its AS, and referring to auxiliary modules for any required auxiliary entities. There was also a separate module for each *sort of construct*, but, in contrast to the original structure, this module did not combine a particular selection of individual constructs: it merely introduced the syntactic sort itself, some meta-variables ranging over it, and the symbol used for the corresponding semantic function.

An ASD of a particular language was given simply by referring to the modules for the required individual constructs and sorts of constructs; an ASD of a

different language could reuse some of the modules, omit others, and add further modules. It was easy to determine which constructs two languages have in common, simply by comparing the references to the modules.

ASF+SDF was used for writing ASDs with the new modular structure [12], and a demonstration involving a small case study has been given [18].

Figure 5 shows the SDF module for the sort *Exp* from [12]. It introduces the semantic function *evaluate* and meta-variables ranging over *Exp*. It imports the auxiliary module *Values*, shown in Fig. 6, which introduces the sort *Value* of expressible data (without constraining it at all). The *Values* module also imports the module *AN*, which introduces the sort *Action* and all the rest of the standard notation for actions. [2]

```
module Exp
  imports Values
  exports
    sorts Exp
    context-free syntax
       "evaluate" "[[" Exp "]]" -> Action %% giving Value
    variables
       "E"[1-9]? -> Exp
```

Fig. 5. Module Exp in SDF

```
module Values
  imports AN
  exports
    sorts Value
    context-free syntax
       Value -> Datum
```

Fig. 6. Module Values in SDF

Figure 7 shows an ASF+SDF module for an ASD of the usual conditional expression, where the condition is supposed to be boolean-valued. It uses SDF to introduce the mixfix notation used for the syntax of the construct, and to require *Bool* to be included in the sort *Value*. It is necessary to import modules for all the sorts of constructs involved in the described construct – here, just *Exp*.

The **equations** part gives an equation in ASF to define the action semantics of the construct, using the notation introduced in the SDF part of the same module and that originating in imported modules. The design of the Action Environment is largely independent of the details of AN: the crucial feature is that when modules are combined to provide an ASD of a complete language,

[2] The version of AN used by Doh and Mosses did not include formal notation for subsorts of *Action*, and the '`giving` *Value*' in Fig. 5 is merely a comment.

```
module Exp/if-then-else
  imports Exp
  exports
    context-free syntax
      "if" Exp "then" Exp "else" Exp -> Exp
      Bool -> Value

equations

[] evaluate [[ if E1 then E2 else E3 ]] =
      evaluate [[E1]] then
      maybe check the boolean then
      evaluate [[E2]] else
      evaluate [[E3]]
```

Fig. 7. Module Exp/if-then-else in ASF+SDF

there should be no need to reformulate any of the actions given in the semantic equations.

For instance, if the conditional expression construct of Fig. 7 were to be included in a language with constructs that allowed expressions to have side-effects (or even spawn threads), no changes would be required. This is possible because action combinators are defined on all possible actions. An action such as $A_1$ `then` $A_2$ makes the data given on normal termination of $A_1$ available to $A_2$, but fails or throws an exception if $A_1$ does that. By the way, the (compound) action `maybe check the boolean` fails when the data given to it is not simply `true`, and $A_1$ `else` $A_2$ performs $A_2$ only when $A_1$ fails. (A more detailed informal introduction to an earlier version of AN is given by Doh and Mosses [12].)

## 4 ASDF

ASDF is a language specification formalism designed to make it easier to write ASDs of single language constructs.

### *4.1 Formalism*

We have previously used plain ASF+SDF for writing ASDs, as described in Section 3. An advantage of using ASF+SDF was that it allowed ASDs to be prototyped using the Meta-Environment. Furthermore other tools, like an action interpreter, action type-checker, etc., could be connected to the Meta-Environment. However, using ASF+SDF for writing small modules describing

single language constructs was not optimal, and this prompted the development of ASDF. The main problems with using ASF+SDF were related to the cumbersome notation:

- When using a syntactic sort, e.g., *Exp*, in a production rule, the module introducing the syntactic sort had to be explicitly imported (see Figure 7). Also modules describing AN had to be imported, since it was not part of the SDF language.
- The declaration of metavariables ranging over sorts is somewhat tedious (see Figure 5).
- ASF+SDF requires many keywords and can be misleading, e.g., the signature of a semantic function is introduced by the words '**context-free syntax**'.

ASDF solves these problems, making specifications easier both to write and read.

```
Exp  ::=  Ide | if Exp then Exp else Exp |
          Exp Exp | fn Ide => Exp
Dec  ::=  val Ide = Exp | Dec Dec
```

Fig. 8. Small subset of ML

**module** *SmallML*

**imports**

*Exp/Ide*   *Exp/Cond*   *Exp/App-Seq*
*Exp/Abs*   *Dec/Bind-Val*   *Dec/Accum*

Module 1

**module** *Exp*

**requires**

*E* : *Exp*

*Datum* `::=` *Val*

**semantics** *evaluate*: *Exp* `->` *Action* `&` *using ()* `&` *giving val*

Module 2

A semantic description of a language consists of a collection of ASDF modules, together with a mapping from the concrete syntax used in the language to the abstract syntax described in the modules. Figure 8 shows a small subset of

**module** *Exp/Ide*

**syntax** *Exp* ::= val(*Ide*)

**semantics** *evaluate* val(*I*) = give the val bound-to *I*

Module 3

**module** *Exp/Cond*

**syntax** *Exp* ::= cond(*Exp*, *Exp*, *Exp*)

**requires** *Val* ::= *Boolean*

**semantics**

```
evaluate cond(E1, E2, E3) = evaluate E1 then
                            maybe check the boolean then
                            evaluate E2 else evaluate E3
```

Module 4

**module** *Exp/App-Seq*

**syntax** *Exp* ::= app-seq(*Exp*, *Exp*)

**requires** *Val* ::= *Func* | func-no-apply

**semantics**

```
evaluate app-seq(E1,E2) =
   evaluate E1 and-then evaluate E2
   then (apply(action(the func#1), the val#2)
         else (throw func-no-apply))
```

Module 5

**module** *Exp/Abs*

**syntax** *Exp* ::= abs(*Ide*, *Exp*)

**requires** *Val* ::= *Func*

**semantics**

```
evaluate abs(I, E) =
    give func(closure(furthermore bind(I, the val)
                      scope evaluate E))
```

Module 6

**module** *Dec*

**requires**

*D* : *Dec*

*Datum* `::=` *Bindings*

**semantics** *declare* : *Dec* `->` *Action* `&` *using ()* `&` *giving bindings*

Module 7

**module** *Dec/Bind-Val*

**syntax** *Dec* `::=` `bind-val(`*Ide*`,` *Exp*`)`

**semantics**

*declare* `bind-val(`*I*`,` *E*`)` `=` *evaluate* *E* `then bind(`*I*`, the val)`

Module 8

**module** *Dec/Accum*

**syntax** *Dec* `::=` `accum(`*Dec*`+)`

**semantics**

*declare* `accum(`*D*`)` `=` *declare* *D*

*declare* `accum(`*D* *D*`+)` `=` *declare* *D* `before` *declare* `accum(`*D*`+)`

Module 9

**module** *Data/Func*

**requires** *Func* `::=` `func(action:` *Action*`)`

Module 10

ML and Modules 2 – 10 can be used to describe the abstract constructs found in the ML subset. The import-relation between the modules can be seen in the screenshot in Figure 9, where the modules at one level import the modules on the lower level, if there is an edge connecting them.

Comparing Module 4 with the module found in Figure 7, one immediately notices that we use abstract syntax with prefix constructors instead of concrete syntax, when describing constructs in ASDF. The advantage of using language independent prefix constructors for abstract syntax is greater reusability. For

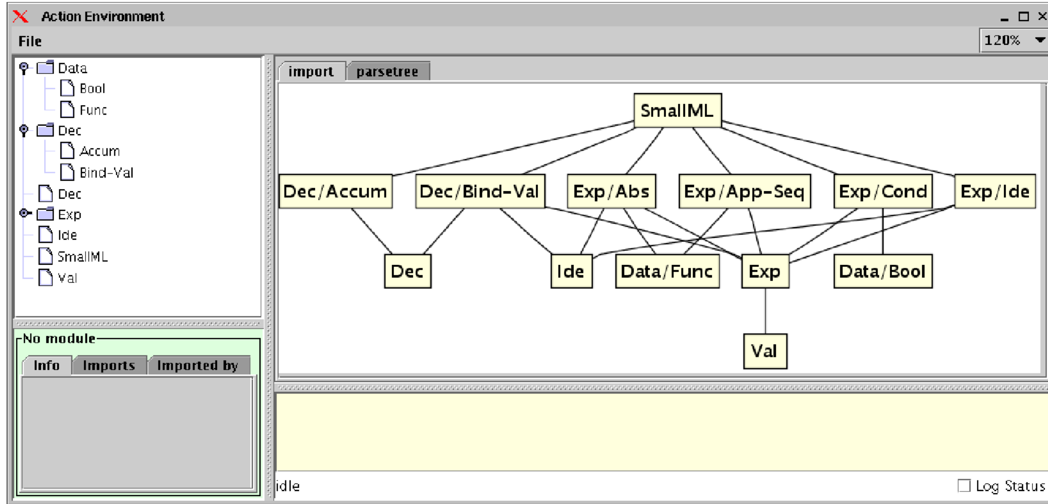
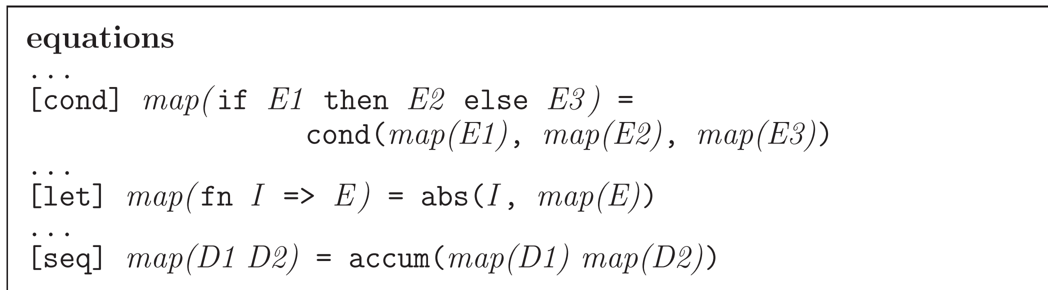



Fig. 9. The Action Environment



**equations**

```
...
[cond] map(if E1 then E2 else E3) =
                  cond(map(E1), map(E2), map(E3))
...
[let] map(fn I => E) = abs(I, map(E))
...
[seq] map(D1 D2) = accum(map(D1) map(D2))
```

Fig. 10. Mapping concrete to abstract syntax

instance, a description of the if-then-else expression from Standard ML might be reused for describing the '?:' expression in Java, since they have the same compositional structure and intended interpretation even though their concrete syntax differs. Part of the definition of the function *map* that maps concrete ML syntax to abstract syntax is shown in Figure 10.

An ASDF module consists of a name (after the keyword **module**) and three optional sections. The **syntax** section defines the abstract syntax of the construct. This is illustrated in Module 3 with the identifier expression constructor `val`, which takes an Identifier (*Ide*) as argument. When writing production rules the familiar separator '`::=`' is used, instead of the '$\longrightarrow$' found in SDF.

The **requires** section is used for introducing data, types, operators, and variables used in the **semantics** section. This is illustrated in Module 6, where the sort *Val* is extended with the sort *Func*, such that actions can produce functions. The syntax for declaring variables is illustrated in Module 7, where '$D$ : *Dec*' declares the variable $D$ to range over the syntactic sort *Dec*. When declaring the variable $X$ to range over a sort $S$ the variables $Xn$, $X*$, and $X+$, where $n$ is a positive integer, are automatically declared to range over the sorts $S$, $S^*$, and $S^+$. The use of these variables is illustrated in Module 4 and Module 9.

Module 10 illustrates how types and operators are introduced. The declaration '*Func* `::= func(action:` *Action*`)`', introduces the type *Func*, and the data operators `func` and `action` becoming available in actions, such that we can write actions as '`give the func`' and '`give action(...)`' (the syntax of the action `give` is '`give` *DataOp*', where *DataOp* contains among other terms '`the` *ANType*'). The operator `func` is a data constructor, and `action` selects the action component of such data.

The semantic function, mapping the abstract syntax construct introduced in the **syntax** section to an action, is defined, using an equation, in the **semantics** section. In the equation, terms from AN and imported modules can be used. For instance, in Module 5 the semantic function contains action combinators and constants, together with the value `func-no-apply` from the **requires** section, and the type `Func` declared in the implicitly imported module *Data/Func*. Notice that it is possible to define the function using more than one equation, as illustrated in Module 9. The **semantics** section can also contain the signature of a semantic function, as we see in Modules 2 and 7. It is required that the signature of a function, used in a module, is defined in the same module or an imported module. The notation used in a semantic equation (besides AN) is defined in the **syntax** and **requires** sections of the module and the imported modules. Therefore parsing a module must be done in two steps, where the first step builds a parsetable based on the **syntax** and **requires** sections. More about this in Sections 4.2 and 5.2.

Syntactic sorts used in the **syntax** section result in implicit imports, so for instance in Module 8 the modules *Dec* (Module 7), *Ide* (not shown), and *Exp* (Module 2) are automatically imported. Implicit imports are also generated from the sorts used in the **requires** section, with the difference that only syntactic sorts used on the right hand side of the production results in imports, and the imported modules always start with *Data/*, for instance Module 6 imports *Data/Func* (Module 10). The automatically imported modules, like *Exp* or *Data/Func*, may provide further sorts than those that caused their importation.

ASDF also allows explicit imports. This is mostly used in the top module that imports all the modules used to describe a language (see Module 1).

The modules presented in this section are simplified versions of the modules used in a semantic description of core ML, which can be found in [15]. The description contains both ASDF modules and ASF+SDF modules mapping ML concrete syntax to abstract syntax.

## 4.2 Environment

The Action Environment supports working with ASF+SDF and ASDF simultaneously, with the restriction that ASF+SDF modules can import ASDF modules, but not the other way round. If there is a name conflict, i.e., an ASF+SDF module and an ASDF module with the same name, it is solved by using the module with the same type as the module importing the problematic module. Being built on top of the ASF+SDF Meta-Environment, the Action Environment inherits most of its features (described in Section 2.3).

On the surface the differences between the Meta-Environment and the Action Environment seem negligible. Because a module in the module graph can be either an ASDF or an ASF+SDF module, different popup menus will appear over modules of different type. Not all features available for ASF+SDF are available for ASDF because they have not yet been implemented (e.g., changing module name and imports). When editing an ASDF module one notices more differences, since the syntax directed editor now uses an ASDF grammar for parsing. Furthermore, the grammar defined in a module (and in the modules it imports) is used when parsing the semantic equations in a module (remember that the equations and the rest of the ASDF module is in the same file, and not in two files as in ASF+SDF). This has two advantages: It gives a better syntactic check of the semantic equation, and it allows the syntax directed structure editor to display the right sorts for the tokens in the semantic equations. As in the Meta-Environment, it is possible to employ the given language specification for parsing and rewriting terms over the language. Due to the way we implemented the Action Environment, everything concerning terms works as in the Meta-Environment.

The advantage of supporting both ASF+SDF and ASDF in the Action Environment is that language descriptions in the environment can describe both concrete syntax (using SDF), abstract syntax constructs, and their semantics (using ASDF), and a mapping from concrete syntax to abstract syntax (using ASF). Using the Action Environment and a description of a language $L$, we obtain a tool for mapping a program written in $L$ to an action.

As in the ASF+SDF Meta-Environment, it is possible to save the parse tables generated by the environment for a specification and the parsed equations collected from the ASF files. Saving parse table and equations to files allows parsing and rewriting terms independently of the Action Environment. The saved parse table and equations can be used to construct a front-end for a compiler. Combining this front-end with an action compiler we obtain a compiler for the language described in the specification.

Different external tools have been integrated into the Action Environment. A

type-checker for action semantic functions (see Section 4.3) gives us a better check of the well-formedness of the ASDF modules and thereby the correctness of the ASD of the language. An action interpreter (see Section 4.4) allows us to interpret programs written in the language we are designing. All in all, the Action Environment should provide a particularly useful environment for developing semantic descriptions and documenting the design of programming languages.

### *4.3 ASDF type checker*

When writing semantic descriptions of programming languages, it is convenient to have tools for checking the descriptions. With ASDF we would like to check that the semantic functions result in actions with certain properties. In the Action Environment it is possible to perform a soft type check of the action in a semantic equation defining a semantic function. The user should provide a signature for the semantic function, and the type checker then checks that the semantic equation conforms to the signature.

```
evaluate: Exp -> Action & using () & giving val &
                 infallible & uncreative

declare: Dec -> Action & using () & giving bindings &
                 infallible & uncreative
```

Fig. 11. Signatures

In Fig. 11 two signatures are displayed. The first describes a semantic function *evaluate* that maps expressions to actions. The actions expect no data (the empty tuple), produce a value, do not fail, and do not create any new memory cells. The second signature, *declare*, describes a mapping to actions with almost the same properties; the only difference is that the actions produce bindings instead of values. When using the first signature to type check the semantic equation in Module 4, the signature is both used to infer the type for the applications of *evaluate* to the sub-expressions and to define the type that we expect the action to have. Notice that a sub-action (`maybe ...`) can fail, but that the failure is caught by the `else` action combinator, so the whole action will not fail. If an action has the type *uncreative* none of its sub-actions must create memory cells. This means that the action cannot contain the action `create`, and the semantic functions occurring in the action must produce actions with type *uncreative*. The action in Module 4 satisfies these two conditions. Module 8 contains an example of a semantic equation where two signatures are needed to type check the action.

Besides the signatures, the type checker also uses other type information from the module containing the semantic equation and modules imported from this

module, for instance, in Module 4 the line "*Val* ::= *Boolean*" defines booleans to be a subtype of values.

Type checking will either result in an error message indicating what might be wrong in the action, or a message saying that the action type checked without problems. Because this is a soft type check, the purpose is not to guarantee that the actions resulting from applying the semantic functions are type correct. Instead the purpose is to warn the language designer against possible problems in the specification. The lack of information about the type of the current bindings in a semantic equation is the reason the type checking is soft. The type of the current bindings can not be computed because the actual bound identifiers are not known in the semantic equations (instead the equations use variables to range over identifiers, see Modules 3 and 8).

### *4.4 Action Interpreter*

An editor buffer containing an action, e.g., the result of applying a semantic function to a program, can be interpreted using the action interpreter connected to the environment. The result of interpreting an action is an indication of how it terminated (normally, abruptly, or failing), the data it produced (if any), and a structure describing the effects evaluating the action has had on storage. The interpreter uses information from the module the action term was opened over and the modules imported from this module. Information about subtype relations, data constructors and selectors, and data constants is used.

## 5 Implementation Overview

The Action Environment is built on top of the ASF+SDF Meta-Environment. Discussing the implementation details of the Action Environment involves discussing the architecture of the Meta-Environment.

### *5.1 ASF+SDF Meta-Environment Architecture*

The Meta-Environment has a layered architecture as displayed in Figure 12. In this section we will discuss each of these layers in more detail. The first step towards a layered design of the ASF+SDF Meta-Environment is discussed in [6]. That paper discusses how ASF can be replaced by another rewriting formalism. This development has been taken a step further, resulting in the architecture discussed here.

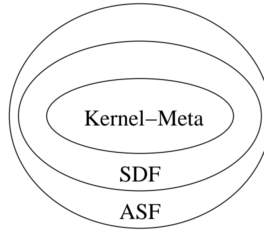



Fig. 12. The layered architecture of the ASF+SDF Meta-Environment

#### *5.1.1 Kernel layer*

The kernel of the Meta-Environment is completely language independent. It consists of the software coordination architecture, the ToolBus [3], which takes care of all the communication between the components that make up the Meta-Environment. The ToolBus allows a full separation of coordination and computation, it is a programmable software bus where the coordination between the components is formally described using a Process Algebra based formalism. The computation is performed within the connected components, which can be implemented in any programming language. The exchange of data between the components is based on a representation format, ATerms [4], specially designed for representing tree-like data structures. This formalism provides maximal subterm sharing and efficient linearization operations.

Besides the ToolBus the kernel of the Meta-Environment consists of a parser, text and structure editors, graphical user interface components, a term store to store parse tables and parse trees, a component which takes care of the communication with the file system, etc. Each of the components is fully language independent and will be instantiated via the next layer, which provides language specific functionality. The kernel is fully prepared to deal with modular languages and specification formalisms.

#### *5.1.2 SDF layer*

The next layer instantiates the kernel Meta-Environment with SDF functionality. This is achieved by adding SDF-specific components to the kernel and by adding actions, via buttons and clickable icons in the user interface, to activate editors for SDF modules. Examples of SDF-specific components are the SDF parse table, the import relation calculator, and the parse table generator. The latter is needed because of the fact that SDF is designed to describe syntax of programming languages, and in order to use these language descriptions it is necessary to generate parse tables for parsing programs. Furthermore, the term store has to be instantiated in such a way that both the parse trees of SDF modules and their corresponding parse tables can be stored.

### *5.1.3 ASF layer*

This layer extends the SDF Meta-Environment with ASF functionality. Again this is achieved by adding ASF-specific components and actions to activate for instance editors for ASF modules. An example of an ASF-specific component is a component which extends every SDF specification with the syntax rules to parse the ASF equations; in this way the user defined syntax in the equations is obtained. Using SDF in combination with ASF poses some restrictions on the grammar rules one can write in SDF, e.g., the separator in a list may only be a literal and not an arbitrary symbol. These restrictions are checked by an ASF+SDF-syntax-checker. Finally, this layer provides an ASF checker to check the well-formedness of the equations, and an ASF interpreter and compiler are added to the SDF Meta-Environment. The term store has to be extended to store ASF modules, corresponding parse tables, etc., as well.

### *5.1.4 Implementation*

Figure 13 shows an abstraction of the kernel Meta-Environment with each of the extensions described above. In this section we will briefly describe how we achieve these extensions in a flexible way.

The messages that can be received by the kernel layer are known in advance, simply because this part of the system is fixed. The reverse is not true: the generic part can make no assumptions about the functionality provided by the other layers.

We identify messages that are sent from the kernel of the Meta-Environment to the extensions as so-called *hooks*. The SDF layer can and will introduce new hooks for the next layers. Each instance of the environment should *at least* implement a receiver for each of these hooks. Implementing these hooks involves writing small pieces of ToolBus specifications. Table 1 shows a few kernel hooks. They are all related to the GUI and editors. The dashed arrows in the Figure 13 between the kernel layer and the ASF or SDF layer denote the hooks and the service requests.

Adding a layer involves some implementation effort. Of course, the components themselves have to be implemented. In a number of cases it is necessary to write ToolBus scripts, but the kernel Meta-Environment also provides a powerful *button language*, which can be used to connect new components and functionality. The button language enables a flexible way of adding buttons and icons to the GUI and adding buttons to the various types of editors.

| Hook | Description |
|---|---|
| `environment-name(Name)` | The main GUI window will display this name |
| `extensions(Sig, Sem, Term)` | Declares the extensions of different file types |
| `stdlib-path(Path)` | Sets the path to a standard library |
| `top-sort(Sort)` | Declares the top non-terminal of a specification |

Table 1
The Meta-Environment hooks: hooks that parameterise the GUI

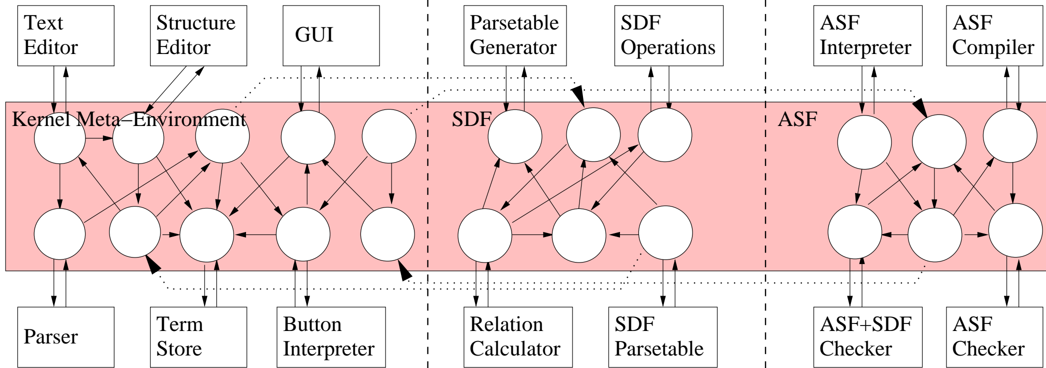



Fig. 13. The layered implementation of the ASF+SDF Meta-Environment

### *5.2 The Action Environment*

In the Action Environment the layered design of the Meta-Environment is extended with an extra layer, the ASDF layer, illustrated in Figure 14. Notice that we do not replace any parts of the ASF+SDF Meta-Environment, we just extend it with an extra layer on the top. In [6] it is described how an environment for another rewriting formalism is implemented by replacing the ASF layer with a layer for the new formalism. This approach is not possible for us because the Action Environment should still support ASF+SDF modules. Another way of viewing the ASDF layer is as an ASDF interface to the ASF+SDF Meta-Environment.

The ASDF layer consists of several components: an ASDF parser, tools for retrieving the module name and imported modules from an ASDF module, and two ASDF to ASF+SDF mappings. As with the other layers we also have to extend the term store, in this case to hold ASDF modules. Based on the grammar of the ASDF language, a parse table has been generated, which is used in the ASDF parser. The tools for getting the module name and imported modules from an ASDF module are implemented in ASF+SDF and are almost trivial (this is the *ASDF Support* component in the illustration). Here we shall focus on the generation of ASF+SDF, and how we have connected external tools.

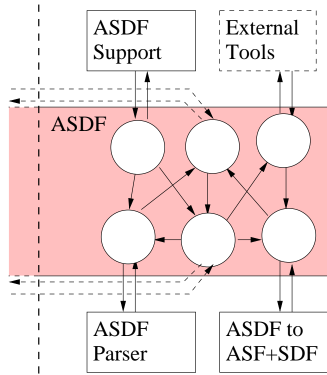



Fig. 14. The ASDF layer

To measure the size of the ASDF layer we have counted the number of ToolBus script lines to approximately 2300 lines compared to approximately 10000 lines in the ASF+SDF Meta-Environment. The tools in the ASDF layer is implemented using approximately 7000 lines of ASF+SDF.

### *5.2.1 Mapping ASDF to ASF+SDF*

The Action Environment contains two mappings of ASDF to ASF+SDF. The result of one mapping is used for parsing and rewriting terms. By mapping every ASDF module to an ASF+SDF module we get the same effect, with respect to working with terms, as if we had opened the generated ASF+SDF modules in the Meta-Environment, so editing of terms is independent of the ASDF layer. The result of the other mapping is used for the second parse of the ASDF module itself (the parse that checks the semantic equations using the notation introduced in the same module and modules imported from it).

We shall use some of the modules in Section 4.1 as examples in this section. The ASDF module's name declaration together with its import section (if any) can be copied verbatim into the ASF+SDF module as illustrated in Fig. 15. Together with the explicit imports from the ASDF module, the generated ASF+SDF also contains imports of modules describing action notation (the module *AN*) and layout characters (the module *Layout*). Implicit imports, as explained in Section 4.1, are translated to explicit imports, e.g., Module 5 uses the sorts *Exp* and *Func*, and the SDF generated from this module imports the modules *Exp* and *Data/Func* (Fig. 17).

The rest of an ASDF module is translated into ASF equations and SDF sections declaring start symbols, sorts, lexical and context-free syntax productions, and variables. The sort declaring sections ensure that all sorts occurring on the right hand side of the arrow in a syntactic function are declared. Examples of this can be seen in Fig. 16 where the sorts *Exp*, *Datum*, *AN-Type*, and *Action* are declared.

```
module SmallML

imports Exp/Ide   Exp/Cond
        Exp/App-Seq   Exp/Abs
        Dec/Bind-Val  Dec/Accum

imports AN   Layout
```

Fig. 15. SDF generated from Module 1

```
module Exp

imports
  AN   Layout   Data/Val

exports
  sorts Exp

  variables
    "E"[0-9]*       -> Exp
    "E"[0-9]*"+"    -> Exp+
    "E"[0-9]*"*"    -> Exp*

  sorts Datum  AN-Type

  context-free syntax
    Val -> Datum

  lexical syntax
    "datum" -> AN-Type
    "val"   -> AN-Type

  sorts Action

  context-free syntax
    evaluate Exp -> Action
```

Fig. 16. SDF generated from Module 2

The sorts which are also declared to be context-free start symbols can be used as the top sort in a parse tree for a term. All sorts defined in the syntax section of an ASDF module are declared to be start-symbols (see Fig. 17).

A production of the type "*Sort* ::= *Symbols*" in the **syntax** and **requires** sections is mapped into a context-free syntax section containing a function "*Symbols* → *Sort*", as shown in Fig. 17. The productions in **requires** sections also result in declaration of types for use in action notation, e.g., in Fig. 16 the production "*Datum* ::= *Val*" is translated to SDF that declares `datum` and `val` to be types for use in action notation.

Module 2 declares variables with the prefix `E`, and this is translated to the

variables section shown in Fig. 16. Here regular expressions over character-sets and strings are used to define variables ranging over *Exp*, *Exp**, and *Exp+*. The variables are used in the ASF generated from **semantics** sections, as shown in Fig. 18. In the semantics section it is only the equations, and not the signatures, that are translated to ASF. The signatures are translated to a syntactic function as shown at the bottom of Fig. 17.

```
module Exp/App-Seq

imports
  AN          Layout
  Data/Func   Exp

exports
  context-free start-symbols Exp

  sorts Exp

  context-free syntax
    app-seq(Exp, Exp) -> Exp

  sorts Val AN-Type

  context-free syntax
    Func                 -> Val

  lexical syntax
    "val"  -> AN-Type
    "func" -> AN-Type

  lexical syntax
    "func-no-apply" -> Val
```

Fig. 17. SDF generated from Module 5

```
equations

[] evaluate app-seq (E1 ,E2) =
      ( evaluate E1 and-then evaluate E2 then
      ( apply (action(the func#1), the val#2)
        else throw func-no-apply ))
```

Fig. 18. ASF generated from Module 5

The ASF+SDF is generated on demand (i.e., when we need to parse a term or a module), and has to be regenerated for an ASDF module every time the module changes. The mappings to ASF+SDF are implemented in ASF+SDF; this was an obvious choice since an SDF grammar for ASF+SDF already exists, which made it easy to construct a type-safe translation.

### 5.2.2 Integration of external tools

Due to the configurability of the Meta-Environment, it is possible to attach external tools, like an action type-checker or interpreter. This is an easy task using the button language, under the assumption that the tools just take the contents of an editor as input, and return a text string as result.

It becomes more complicated when the tool needs global information (like a semantic function type-checker, which needs all imported function signatures to check a function definition), and in these cases we need to traverse the import graph to collect the necessary information from each module.

```
action([description(asdf-editor,
                    menu(["Actions", "Type check"]))],
       [push-active-module,
        prompt-for-file("Extra type constraints", "",".asdf"),
        split-file-name,
        type-check-asdf])
```

Fig. 19. Definition of type checking menu item

Figure 19 shows the definition of the menu item that starts the ASDF type checker written in button language. In the Meta-Environment anything the user can click is referred to as a button, hence also a menu item. The first line defines where the button should occur, and in this case it only occurs in ASDF editors. The second line describe how it should occur, and in this case it occurs as a menu item named “Type check” under the menu “Actions”. The rest of the lines define the buttons behavior, using a special stack based script language. The command `push-active-module` pushes the name of the module in the editor on the stack, before the command `prompt-for-file` asks the user for an ASDF file containing extra type information. Using the stack, the name of the file is passed to the next command (`split-file-name`) which splits the file name into directory, name, and extension. Finally the command `type-check-asdf` calls the ToolBus interface to the type checker.

## 6 Related Work

An enormous amount of work has been performed in the field of defining the syntax and semantics of programming languages and systems supporting the development of such language definitions. We refer to Heering and Klint [14] for a fairly complete and up-to-date overview.

In the discussion of related work we will focus on environments which can be used to describe single language constructs in a modular way, or to give ASDs of languages.

The GEM-MEX system [1] allows description of languages using a collection of MONTAGES, a formalism based on Abstract State Machines. The idea of describing single language constructs in separate modules is encouraged by GEM-MEX, but due to the lacking modularity of the syntax formalism used (the semantic descriptions of individual constructs are based on concrete syntax, and the collected syntax has to be LALR(1)) a MONTAGE is not often reusable in practice.

The ABACO system [21] is an AS tool for students and programming language designers. The main components of ABACO are an algebraic specification compiler, specification editors, action libraries, action editors, and a GUI. Furthermore, it offers a help system, an action debugger and facilities to export specifications to readable output. The main component is the algebraic specification compiler, which provides syntax checking of specifications and interpretation. The ABACO system and the Action Environment have a strong resemblance, but the Action Environment offers more flexibility in adding external components by means of openness of the underlying architecture.

The action semantics of individual constructs can be presented with an object-oriented perspective [9]. Then the introduction of each syntactic sort and its corresponding semantic function is given as a class definition; the syntax of an individual construct and its action semantics are defined in a subclass that extends the class defining the sort of the construct. The use of conventional object-oriented class definitions does not allow as much to be left implicit as in ASDF, but otherwise the collections of class and subclass definitions are directly comparable to collections of modules in ASDF. However, tool support for the approach has not yet been provided.

The ASD toolset [10] supported the creation, editing, checking, and use of ASDs. This toolset had a very strong relation with an older version of ASF+SDF, and its implementation has become obsolete.

## 7 Conclusions and Future Work

In this paper, we have presented ASDF, a new formalism for action semantic descriptions supporting reuse of descriptions of individual constructs. We have also reported on the Action Environment, a new environment supporting the use of ASDF, and explained how it is implemented on top of the ASF+SDF Meta-Environment. Two of the authors have already carried out an initial case study in the use of ASDF and the Action Environment, providing ASDF modules for all the basic constructs underlying Core ML (accepted for publication in IEE Proceedings Software).

Plans for future work include further case studies in the use of ASDF, and improving the action interpreter and the ASDF type checker.

**Acknowledgements** Thanks to Janus Dam Nielsen for doing a great job in improving the Action Environment. The SEN1 group at CWI, specially Jurgen Vinju and Hayco de Jong, have been very helpful in providing technical information about the ASF+SDF Meta-Environment.

## References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In *PSI'99*, LNCS Vol. 1755, pages 40–53. Springer, 2000.

[2] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. Addison-Wesley, 1989.

[3] J. A. Bergstra and P. Klint. The discrete time ToolBus – A software coordination architecture. *Sci. Comput. Programming*, 31(2-3):205–229, 1998.

[4] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[5] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. In *LDTA 2004*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[6] M. G. J. van den Brand, P. Moreau, and J. J. Vinju. Environments for term rewriting engines for free! In *RTA 2003*, LNCS Vol. 2706, pages 424–435. Springer, 2003.

[7] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In *CC 2001*, LNCS Vol. 2027, pages 365–370. Springer, 2001.

[8] D. Brown and D. A. Watt. JAS: A Java action semantics. In *AS'99*, BRICS NS-99-3, pages 43–55. Dept. of Computer Science, Univ. of Aarhus, 1999.

[9] C. Carvilhe and M. Musicante. An object-oriented view of action semantics. In *AS 2002*, BRICS NS-02-8, pages 45–64. Dept. of Computer Science, Univ. of Aarhus, 2002.

[10] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, Univ. of Amsterdam, 1994.

[11] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping:*

*An Algebraic Specification Approach*. AMAST Series in Computing Vol. 5. World Scientific, 1996.

[12] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Programming*, 47(1):3–36, 2003.

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

[14] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.

[15] J. Iversen and P. D. Mosses. Constructive action semantics for core ML. *IEE Proceedings-Software spec. issue on Language Definitions and Tool Generation*, 2004. To appear.

[16] 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*, BRICS NS-00-6, pages 19–36. Dept. of Comput. Sci., Univ. of Aarhus, 2000.

[17] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96*, LNCS Vol. 1058, pages 219–234. Springer, 1996.

[18] P. Mosses. Action Semantics and ASF+SDF. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[19] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[20] P. D. Mosses. A modular SOS for Action Notation. BRICS RS-99-56, Dept. of Comput. Sci., Univ. of Aarhus, 1999.

[21] H. Moura, L. C. Menezes, M. Monteiro, P. Sampaio, and W. Cansanção. The Abaco system: An action tool for programming language designers. In *AS 2002*, BRICS NS-02-8, pages 1–8. Dept. of Computer Science, Univ. of Aarhus, 2002.

[22] D. A. Watt. The static and dynamic semantics of SML. In *AS'99*, BRICS NS-99-3, pages 155–172. Dept. of Computer Science, Univ. of Aarhus, 1999.

# Recent BRICS Report Series Publications

**RS-04-36** **Mark van den Brand, Jørgen Iversen, and Peter D. Mosses.** ***An Action Environment*. December 2004. 27 pp. Appears in Hedin and Van Wyk, editors, *Fourth ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, LDTA '04, 2004, pages 149–168.**

**RS-04-35** **Jørgen Iversen.** ***Type Checking Semantic Functions in ASDF*. December 2004.**

**RS-04-34** **Anders Møller and Michael I. Schwartzbach.** ***The Design Space of Type Checkers for XML Transformation Languages*. December 2004. 21 pp. Appears in Eiter and Libkin, editors, *Database Theory: 10th International Conference*, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.**

**RS-04-33** **Aske Simon Christensen, Christian Kirkegaard, and Anders Møller.** ***A Runtime System for XML Transformations in Java*. December 2004. 15 pp. Appears in Bellahsene, Milo, Rys, Suciu and Unland, editors, *Database and XML Technologies: Second International XML Database Symposium*, XSym '04 Proceedings, LNCS 3186, 2004, pages 143–157. Supersedes the earlier BRICS report RS-03-29.**

**RS-04-32** **Philipp Gerhardy.** ***A Quantitative Version of Kirk's Fixed Point Theorem for Asymptotic Contractions*. December 2004. 9 pp.**

**RS-04-31** **Philipp Gerhardy and Ulrich Kohlenbach.** ***Strongly Uniform Bounds from Semi-Constructive Proofs*. December 2004. 31 pp.**

**RS-04-30** **Olivier Danvy.** ***From Reduction-Based to Reduction-Free Normalization*. December 2004. 27 pp. Invited talk at the *4th International Workshop on Reduction Strategies in Rewriting and Programming*, WRS 2004 (Aachen, Germany, June 2, 2004). To appear in ENTCS.**

**RS-04-29** **Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy.** ***An Operational Foundation for Delimited Continuations in the CPS Hierarchy*. December 2004. iii+45 pp.**

**RS-04-28** **Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*. December 2004. 44 pp. Extended version of an article to appear in *Theoretical Computer Science*.**