BRICS RS-04-37 Iversen & Mosses: Constructive Action Semantics for Core ML

BRICS

Basic Research in Computer Science

Constructive Action Semantics for Core ML

Jørgen Iversen
Peter D. Mosses

BRICS Report Series **RS-04-37**

ISSN 0909-0878 **December 2004**

Constructive Action Semantics for Core ML

Jørgen Iversen, Peter D. Mosses

BRICS* & Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark
j.iversen@brics.dk, pdmosses@brics.dk

December, 2004

Abstract

Usually, the majority of language constructs found in a programming language can also be found in many other languages, because language design is based on reuse. This should be reflected in the way we give semantics to programming languages. It can be achieved by making a language description consist of a collection of modules, each defining a single language construct. The description of a single language construct should be language independent, so that it can be reused in other descriptions without any changes. We call a language description framework "constructive" when it supports independent description of individual constructs.

We present a case study in constructive semantic description. The case study is a description of Core ML, consisting of a mapping from it to BAS (Basic Abstract Syntax) and action semantic descriptions of the individual BAS constructs. The latter are written in ASDF (Action Semantics Definition Formalism), a formalism specially designed for writing action semantic descriptions of single language constructs.

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

Tool support is provided by the ASF+SDF Meta-Environment and by the Action Environment, which is a new extension of the ASF+SDF Meta-Environment.

1 Introduction

We will present a case study in the use of a recently-developed approach to modular definition of programming languages, called *Constructive Action Semantics*.[1] The case study consists of a description of the dynamic semantics of Core ML (the sublanguage of Standard ML [2] obtained by removing the constructs concerning modules). We have chosen Core ML for this case study primarily because it is a cleanly-designed, medium-size language, and our description of it provides substantial evidence that our approach is practical.

The main point of our approach, amply demonstrated by the case study, is that a description of a complete language should be based on reusable definitions of *individual language features*. This requires an extremely high degree of modularity, and ensures extensibility and modifiability. Moreover, it significantly lowers the investment of effort required to give a semantic description of a new language, since no re-description of constructs taken from previously-described languages is needed. We hope that this will encourage language designers to use formal semantics for documentation purposes to a much greater extent than at present [3, 4].

An important feature of our approach is the tool support available for it: the so-called Action Environment, which is implemented as an extension of the ASF+SDF Meta-Environment [5, 6]. The Action Environment is still at an early stage of development, and it does not yet provide full functionality regarding prototyping, nor other tool generation; nevertheless, we have found the present version highly useful in connection with the case study, and we include a brief description of the main features here.

The rest of this section explains and motivates Constructive Action Semantics, mentions related work, and gives an overview of the rest of the paper.

[1]Constructive Action Semantics was referred to as "Incremental" in [1], but a reviewer found the terminology confusing.

Motivation

Although each programming language generally has some quite distinctive features, it usually also has many features in common with other languages. This is particularly true for languages in the same family (e.g., C, C++, and Java), and those related as extensions (e.g., Standard ML and Concurrent ML [7]). Moreover, some features seem to be almost ubiquitous: examples include sequencing, conditional, iteration, assignment, procedural abstraction, and local declarations. There are sometimes differences regarding details of both the form and intended interpretation of such a feature (e.g., how conditional expressions are written, and whether the conditions are numerical or boolean), but if we take a 'feature' to be an individual *abstract* construct together with a *particular* intended interpretation for it, that exact feature may often be found in a significant number of different languages.

Typically, a semantic description of a particular feature is given only in connection with some complete language description, and common features thus get re-described many times. In most semantic frameworks (including the conventional styles of denotational and operational semantics, but not action semantics) the description of each feature depends critically on what other kinds of features are included in the described language. Even when that is not the case, there may still be substantial notational variation between different descriptions of the same feature. In practice, semantic descriptions of individual features have seldom been reused verbatim.

Let us call a framework *constructive* when it supports the independent definition and use of named modules describing individual language features. We insist on reuse by reference to names of modules, rather than by copying, since in the latter case the reader cannot easily see whether the copy has subsequently been edited or not. Note however that merely introducing explicit modular structure into a semantic description does *not* ensure that it is constructive: the crucial question is whether the bodies of the modules are independent.

The advantages of a constructive framework are both numerous and significant:

- an expanding library of independent descriptions of individual language constructs can be provided;
- when describing a new language, no time and effort is wasted on re-describing familiar constructs;

- when the descriptions of two different languages refer to the same module, it is immediately apparent that they give the same interpretation of the construct that it describes;

- when a module is already familiar from its use in a previous description, the reader can skip it;

- notational variations between different descriptions are minimised; and

- extensibility and modifiability are guaranteed.

Constructive Action Semantics is a constructive version of the original Action Semantics framework [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. Action Semantics is a hybrid of denotational and operational semantics: semantic functions map programs and their phrases compositionally to actions expressed in Action Notation, which is itself defined operationally. The original modular structure of action semantic descriptions was quite conventional, with top-level modules specifying abstract syntax, semantic entities, and semantic functions. The modules for abstract syntax and semantic functions were divided into modules for expressions, statements, declarations, etc., and different sorts of semantic entities were specified in separate modules, in the usual tradition of algebraic specifications of abstract datatypes. Although the semantic equations defining the action semantics of individual constructs were indeed independent of each other, they were not isolated as named modules, and reuse of parts of action semantic descriptions was restricted to copy-and-paste.

The modular structure of a constructive action semantic description is essentially the conventional structure turned inside-out:

- a module for each *sort of construct* in the abstract syntax declares the sort itself, variables ranging over that sort, and a semantic function for that sort, referring to auxiliary modules for any generally-required semantic entities;

- a module for each *individual construct* declares the abstract syntax constructor function and gives the semantic equation defining its action semantics, referring to auxiliary modules for any semantic entities required for that construct.

The modules for the semantic entities are essentially as in the original version of Action Semantics.

The aim of reusability has a significant impact on the specification of abstract syntax. Regarding notation, the conventional style of abstract syntax is strongly suggestive of the concrete syntax of the language being described. This is clearly inappropriate for constructive action semantic descriptions, where the module specifying the abstract syntax and semantics of an individual construct is to be reusable in connection with languages that have significantly different concrete syntax for the same abstract constructs. For example, consider the conditional expression: in ML it is written as '`if...then...else...`', but in Java as '`...?...:...`'. Not wishing to bias our notation toward any particular family of languages, we are forced to adopt prefix notation and use neutral words for abstract syntax constructor functions, e.g., 'cond' for conditional expressions. (We still rely on positional notation for components of constructs, although this might occasionally be a source of potential confusion.)

Moreover, to maximise reusability, we take account of two deeper aspects of the choice of abstract syntax:

- We limit ourselves to a *fixed set of sorts* of constructs, corresponding to the fundamental distinctions between expressions, statements, declarations, parameters, etc. These distinctions reflect differences in the kinds of values computed by constructs and whether or not the constructs may depend on bindings and arguments (e.g., constants and operations are distinguished from expressions). Note that we allow subsort inclusions to be specified in connection with particular languages.

- We include only constructs which are *not easily derivable* from other constructs. For instance, the single-branch conditional statement can be derived from the double-branch conditional one and the null statement, so we omit it; similarly for conditional conjunction, which can be derived from the conditional expression. However, we do not go as far as *The Definition of Standard ML* [18], where for instance the conditional expression is derived from a more general case construct, and tuples are derived from records.

The above principles lead us to a Basic Abstract Syntax (BAS) with a fixed set of sorts but an open-ended set of constructors. The mapping from concrete syntax to BAS is a significant and informative part of a complete language description.

Overview

The rest of the paper proceeds as follows. Section 2 recalls the concrete syntax of Core ML (mainly for the benefit of readers who are unfamiliar with ML). Section 3 explains how the illustrated concrete syntax constructs are mapped to BAS, introducing the sorts and constructs of BAS in the process. Section 4 gives reusable action semantic definitions of the illustrated BAS constructs, introducing the notation used in action semantics in the process. The composition of the mapping from concrete ML syntax to BAS with the action semantics of the BAS constructs provides our constructive action semantics for the ML constructs. The full specifications on which Sects. 2–4 are based are provided in appendices. Section 5 discusses the reusability of the specified modules. Section 6 presents the Action Environment which supported the development of the modules. Section 8 concludes by summarising what has been achieved, and indicates topics for future work.

2 ML Syntax

ML [2] is a strict, functional, polymorphic programming language with exception handling, immutable data types, updatable references, abstract data types, and parametric modules.

In this section we will introduce examples of the concrete syntax of Core ML (i.e., Standard ML excluding modules), to familiarise the reader with the language. We will not be very strict with our description of the ML syntax, and we leave out details which are either irrelevant or excessively cumbersome to describe. Appendix A contains an SDF[2] grammar of the whole Core ML syntax, which is consistent with the grammar found in *The Definition of Standard ML* [18]. In the next sections we will use the constructs introduced in this section as examples when giving a semantics for ML. Readers already familiar with ML might prefer to take a quick look at Table 1 to get an idea of the subset of Core ML whose abstract syntax and semantics we will be describing in the following two sections, and then skip to the next section.

Table 1 is a grammar for the concrete syntax of the subset of Core ML we will describe in this section. The nonterminal *CON* expands to constants like integers or strings. Identifiers, consisting of either alphanumeric characters or symbols like '`:=`', '`+`', etc., are derived from the nonterminal *IDE*.

[2] The Syntax Definition Formalism [19, 20].

EXP	::=	*CON* | *IDE* | *EXP EXP* | *EXP IDE EXP* |
		`if` *EXP* `then` *EXP* `else` *EXP* |
		`let` *DEC* `in` *EXP* `end` | `while` *EXP* `do` *EXP* |
		`fn` *PAT* `=>` *EXP* | `(`*EXP*`; ...;` *EXP*`)` |
		`raise` *EXP* | *EXP* `handle` *PAT* `=>` *EXP* |
		`(`*EXP*`, ...,` *EXP*`)` | `(`*EXP*`)` | `[`*EXP*`, ...,` *EXP*`]`
PAT	::=	`_` | *CON* | *IDE* | `(`*PAT*`, ...,` *PAT*`)` |
		`[`*PAT*`, ...,` *PAT*`]` | *IDE PAT*
DEC	::=	`val` *PAT* `=` *EXP* | `fun` *IDE PAT* `=` *EXP* |
		DEC `;` *DEC* | `local` *DEC* `in` *DEC* `end` |
		`datatype` *IDE* `=` *IDE* `of` *TYP* `| ... |` *IDE* `of` *TYP* |
		`exception` *IDE* `of` *TYP*
TYP	::=	*IDE* | *TYP* `->` *TYP* | *TYP* `*` *TYP*

Table 1: ML Grammar

2.1 Expressions

ML doesn't have statements: expressions are used to describe the behaviour that we would use statements to describe in an imperative language.

In ML the atoms in expressions are constants, e.g., integers and strings, and identifiers bound to values. From these atoms new expressions can be formed, for instance by applying functions to expressions, written as '*EXP EXP*'. As opposed to languages like C or Pascal, function application in ML consists of two expressions: the first expression evaluates to the function (this expression does not have to be an identifier) and the other to the argument. Function application can also be written in infix form, like '*EXP IDE EXP*', where *IDE* is an identifier which has been declared infix, and bound to a binary function.

In ML '`if` *EXP* `then` *EXP* `else` *EXP*' expresses a choice between two alternatives based on a condition, but be aware of the difference from Java (and similar languages) where the if-then-else construct is a choice between two statements, which means that it does not evaluate to a value. These languages use the '`?:`' operation to describe a choice between two expressions.

Most languages have a notion of different kinds of scope of declarations. In C (and similar languages) the curly brackets delimit a local scope, where the declarations given in the beginning are valid till the closing bracket. The construct '`let` *DEC* `in` *EXP* `end`' is ML's way of making declarations local to an expression.

Describing a repetition of a computation can be obtained using the '`while` *EXP* do *EXP*' expression. This construct is similar to the iteration statements found in many imperative languages. It is of course important that the body of the expression (the second *EXP*) has side-effects if the evaluation is ever to terminate.

In ML, writing functions is not restricted to declarations: we can also write anonymous functions, which are not bound to identifiers. These expressions evaluate to a function value and are written '`fn` *PAT* `=>` *EXP*' (the syntactic sort *PAT* is described in the next subsection).

As mentioned earlier, expressions replace statements when we compare ML with many imperative languages. One important construct in imperative languages is a sequence of statements, and since ML has expressions with side effects (based on built-in datatypes and data operations) it is not surprising that ML allows sequences of expressions, which are written '`(`*EXP*`; ...;` *EXP*`)`'.

Exceptions are found in many languages, because they allow the programmer to describe a control flow that would otherwise be difficult to delineate. ML has two constructs related to exceptions: '`raise` *EXP*' throws an exception (comparable to the '`throw`' keyword in Java), and '*EXP* `handle` *PAT* `=>` *EXP*' catches exceptions raised in the first expression (comparable to the '`catch`' keyword in Java).

The set of expressible values in ML contains, among others, tuples and lists. Expressions which evaluate to tuples look like '`(`*EXP*`, ...,` *EXP*`)`', and the syntax for lists is similar, but with square brackets instead of round brackets. Notice that tuples of size one don't exist in ML: '`(`*EXP*`)`' is just used for grouping expressions.

ML contains more expressions than those just mentioned; they can all be found in App. A.1.

2.2 Patterns

An important construct used in both ML expressions and ML declarations is the pattern. A pattern describes a set of values by combining constants, data constructors, and variable identifiers. When matched with a value a pattern generates bindings of the identifiers in the pattern to parts of the value. We might say that whereas expressions construct new values, patterns de-construct them.

In ML the simplest pattern is the wild-card pattern '`_`', which matches everything. Built-in constants (e.g., integers or strings), user defined data constants and variable identifiers can also be used as patterns.

Bigger patterns, like tuples of patterns '`(`*PAT*`, ..., `*PAT*`)`' are also a part of ML, and often used to write a tuple of identifiers as the parameters for a function. List patterns are similar, but use square brackets instead of normal brackets.

ML allows users to define their own data types and data constructors, and it includes corresponding patterns to match constructed data. Writing '*IDE PAT*' matches data constructed by applying the constructor *IDE* to a value which matches *PAT*.

Further details about the syntax of patterns are given in App. A.2.

2.3 Declarations

In ML, all expressible values can be bound to identifiers.

The construct '`val` *PAT* `=` *EXP*' generates bindings of identifiers in the pattern to values computed from the subexpressions of *EXP*; the case where *PAT* is simply an identifier corresponds to a simple constant declaration in other languages. It is not a variable declaration (like '`int i;`' in C) because the bindings cannot be updated. Furthermore, types are not mandatory in ML value declarations, since the intended type can usually be inferred (from *EXP* and the usage of the identifiers in the scope of the declaration).

Recursive functions can be declared by writing '`fun` *IDE PAT* `=` *EXP*', where *PAT* is a pattern describing the formal parameters used in the body expression. In many other languages, the only way of defining parameters for a function is a tuple of identifiers; ML is more general, allowing other kinds of patterns as well (possibly nested).

ML allows sequences of declarations separated by '`;`'. In a previous subsection we introduced declarations which had a scope local to an expression. In ML one can also write declarations which are local to declarations: '`local` *DEC* `in` *DEC* `end`'.

ML has datatype declarations, where a new type with different constructors is introduced. It looks like this:

```
datatype IDE_0 = IDE_1 of TYP_1
               | ...
               | IDE_n of TYP_n
```

where IDE_0 is the name of the new datatype and $IDE_1, \ldots, IDE_n$ are the names of the data constructors. The types $TYP_1, \ldots, TYP_n$ describe the arguments of the data constructors; '`of` TYP_i' is omitted when IDE_i has no arguments.

As mentioned in a previous subsection, ML contains expressions which can raise and handle exceptions. Writing '`exception` *IDE* `of` *TYP*' declares an exception named *IDE* with an argument of type *TYP*. The type is optional, so that one can also define exceptions without arguments.

A full description of the syntax of the declarations in Core ML is available in App. A.3.

2.4 Types

ML is a strongly typed language. The type system consists of basic types which are just type names (for instance declared using the datatype construct from the previous subsection) and constructed types like function types '*TYP* `->` *TYP*' and tuple types '*TYP* `*` *TYP*'. See App. A.4 for the full specification of the syntax of types in Core ML.

2.5 Parsing peculiarities

ML is not a context-free language regarding grouping of expressions, because the user can declare identifiers to be infix operations. The string '`x y z`' illustrates the problem. It can be parsed in different ways depending on whether `y` has been declared infix or not. If not, it is parsed as an application of `x` to `y` and an application of this to `z`, where `x` must be bound to a function taking one argument and giving a function which takes one argument. If on the other hand `y` has been declared infix, it is parsed as an infix expression, and `y` must be bound to a binary function. The problem of constructing the right parse tree can be solved by always parsing '`x y z`' initially as a double function application, and subsequently traversing the parse tree, replacing double applications with infix expressions, depending on the context.

There are other non-context free properties of ML, but they are not relevant for the mapping of ML to BAS, and will therefore not be described here.

3 Reduction to Basic Abstract Syntax

This section gives examples of a mapping from ML constructs to Basic Abstract Syntax (BAS). BAS is an evolving selection of basic constructs from different programming languages, to include all the commonly occurring constructs, as well as more specific ones. In the next section we will give an action semantics of the BAS constructs, thus indirectly providing an action semantics for the ML constructs. The mapping to BAS is described by recursive functions which perform a traversal of the concrete syntax tree while building the BAS tree. The syntax of the BAS constructs is given in App. B and the mapping is described in App. C.

The alternative to mapping ML constructs to BAS constructs is to map ML constructs directly to actions. We claim that introducing BAS as an intermediate level is beneficial, because the BAS constructs can be reused not only within the description of ML, but also in descriptions of other languages.

We are only concerned with the dynamic semantics of ML, and consequently we will not describe a mapping of types to BAS: types are just ignored when mapping the other ML constructs.

BAS is divided into a fixed set of syntactic sorts. Constructs describing expressions belong to the sort *Exp*, common to them is that they evaluate to values. Statements belong to *Stm* and these constructs evaluate to the value *null-val*. Matching values against parameters is described with constructs from *Par*, which compute bindings. The syntactic sort *Dec* contains declarations, which also compute bindings. BAS also contains constants *Con* (included in both *Exp* and *Par*), and identifiers *Ide*.

In this section we shall use meta-variables ranging over the syntactic sorts of ML introduced in Table 1. The variables are $C : CON$, $I : IDE$, $E : EXP$, $D : DEC$, $T : TYP$ and $P : PAT$. We will use the convention that a variable with a superscript $\top$ means the translation to BAS of the variable without the superscript, so for instance $E^\top = exp2bas(E)$, where *exp2bas* is the function mapping constructs in EXP to constructs in *Exp*.

Table 2 shows an example of how ML is mapped to BAS.

3.1 Expressions

The function *exp2bas* is fully described in App. C.1. In this section we shall see some examples of this description. The examples are listed in Table 3.

Constants are included in *Exp*, so the result of applying *exp2bas* to a

```
exp2bas(
  let
    exception Negative;
    fun fac 0 = 1
        | fac n =
            if n > 0 then n * fac (n - 1)
            else raise Negative
  in
    fac 5 handle Negative => 0
  end
)
⇒

local(
  accum(
    bind-val(val-or-var(Negative), new-cons)
    rec(bind-val(var(fac),
      app-seq(abs(var(f), abs(var(vid_0),
          app-seq(val(f), tuple-seq(val(vid_0))))),
        alt-seq(
          abs(tuple(0), 1)
          abs(tuple(val-or-var(n)),
            cond(app-seq(val(>), tuple-seq(val(n) 0)),
              app-seq(val(*),
              tuple-seq(val(n) app-seq(val(fac),
                app-seq(val(-), tuple-seq(val(n) 1))))),
              throw(val(Negative)))
          ))))
    )
  ),
  catch(app-seq(val(fac), 5), abs(val-or-var(Negative), 0))
)
```

Table 2: Mapping ML to BAS

C	$\rightarrow$	C
I	$\rightarrow$	*val*(I)
E_1 E_2	$\rightarrow$	*app-seq*($E_1^\top$, $E_2^\top$)
E_1 I E_2	$\rightarrow$	*app-seq*(*val*(I), *tuple-seq*($E_1^\top$ $E_2^\top$))
`if` E_1 `then` E_2 `else` E_3	$\rightarrow$	*cond*($E_1^\top$, $E_1^\top$, $E_1^\top$)
`let` D `in` E `end`	$\rightarrow$	*local*($D^\top$, $E^\top$)
`while` E_1 `do` E_2	$\rightarrow$	*seq*(*while*($E_1^\top$, *stm*($E_2^\top$)),*null-val*)
`fn` P `=>` E	$\rightarrow$	*abs*($P^\top$, $E^\top$)
`(`E_1`; ... ;`E_{n-1}`;`E_n`)`	$\rightarrow$	*seq*(*seq*(*stm*($E_1^\top$)...*stm*($E_{n-1}^\top$)), $E_n^\top$)
`raise` E	$\rightarrow$	*throw*($E^\top$)
E_1 `handle` P `=>` E_2	$\rightarrow$	*catch*($E_1^\top$, *abs*($P^\top$, $E_2^\top$))
`(`E_1`, ..., `E_n`)`	$\rightarrow$	*tuple-seq*($E_1^\top$ ... $E_n^\top$)
`(`E`)`	$\rightarrow$	$E^\top$
`[`E_1`, ..., `E_n`]`	$\rightarrow$	*app*(*list*, *tuple-seq*($E_1^\top$, ..., $E_n^\top$))

Table 3: ML expressions to BAS mapping

constant is the same constant. Less trivial is the mapping of identifiers, since identifiers might be bound to different sorts in different languages. In imperative languages, identifiers can usually be bound to procedures and memory cells, and this requires a combination of two different interpretations of the BAS construct, *val*(I), representing identifier expressions, depending on what the identifier is bound to. In ML, identifiers can be bound to values, which include integers, strings, functions etc., but they can also be bound to data constructors, in which case the behaviour is a bit different (when used in patterns).

Function application 'E_1 E_2', where E_1 evaluates to a function and E_2 is the argument given to this function, is mapped to *app-seq*($E_1^\top$, $E_2^\top$), which insists on left-to-right evaluation of the subexpressions. BAS also contains the *app* construct, which allows interleaving the evaluation of the two subexpressions, but the expressions in a function application are evaluated sequentially in ML. The construct *app-seq* is also used to describe the infix version of function application 'E_1 I E_2', which becomes *app-seq*(*val*(I), *tuple-seq*($E_1^\top$ $E_2^\top$)). Here we use the *tuple-seq* construct instead of the *tuple* construct for the same reason that we choose the *app-seq* construct. For economy, BAS provides only unary function application, using tuples to represent multiple arguments—this is especially convenient for ML, but arguably appropriate for other languages too.

ML's conditional expression '`if` E_1 `then` E_2 `else` E_3' is mapped to *cond(*$E_1^\top$*,* $E_2^\top$*,* $E_3^\top$*)*. Since $E_1^\top$ is expected to evaluate to either true or false, the mapping is trivial, whereas in languages where $E_1^\top$ should evaluate to an integer equal to zero or not, the mapping would have been a bit more complicated; alternatively, we could use a variant of the *cond* construct where the condition is always numerical.

The construct '`let` D `in` E `end`' is mapped to *local(*$D^\top$*,* $E^\top$*)*, which is overloaded because it can also combine two declarations, as we shall see in Sect. 3.3.

ML's iterative expression, '`while` E_1 `do` E_2', is mapped to *seq(while(*$E_1^\top$*, stm(*$E_2^\top$*)), null-val)*. The reason that it is not just mapped to *while(*$E_1^\top$*,* $E_2^\top$*)* is that it is an expression in ML, therefore it must compute a value (in this case *null-val*, corresponding to ML's '`()`'), so we wrap it in a construct which follows a statement by an expression. Furthermore, the usual *while* construct in BAS expects a statement as its second argument, so we use the *stm* construct to get a statement from an expression by discarding the value.[3]

The anonymous function '`fn` P `=>` E' is mapped to *abs(*$P^\top$*,* $E^\top$*)*, which gives static scopes for free occurrences of identifiers.

BAS has various sequence constructs. In the mapping of ML expression sequences, two different sequence constructs are used: a sequence of any number of statements, and a sequence consisting of a statement followed by an expression. ML's sequence of expressions can be seen as a sequence of statements followed by an expression, because the values computed in the first expressions are thrown away and only their side effects are preserved. Thus we can map the sequence '`(`E_1`; ...;` E_{n-1}`;` E_n`)`' to *seq(seq(stm(*$E_1^\top$*) ... stm(*$E_{n-1}^\top$*)),* $E_n^\top$*)*. Notice that *seq* is overloaded, and used in two different ways in this example.

Raising an exception is mapped to *throw(*$E^\top$*)*. Handling exceptions 'E_1 `handle` P `=>` E_2' is mapped to *catch(*$E_1^\top$*, abs(*$P^\top$*,* $E_2^\top$*))*, where we have used *abs* to describe the function on the right-hand side, which will be applied to the exception raised by $E_1^\top$.

The construct '`(`E_1`, ...,` E_n`)`' has a trivial mapping to *tuple-seq(*$E_1^\top$ ... $E_n^\top$*)*, which implies left-to-right evaluation of subexpressions. ML does not have tuples of size one: brackets around a single expression merely indicates

[3] It is of course not possible to eliminate the *while* construct by syntactic unfolding, as the unfolding process would never terminate.

grouping, and can just be removed in the translation to BAS.

When mapping ML lists '`[`E_1`, ..., `E_n`]`' to BAS, we use the data operation *list*, which maps a tuple value to a list value with the same elements. The result is *app(list, tuple-seq(*$E_1^\top$*, ...,* $E_n^\top$*))*. An alternative mapping would be to use the fact that, according to *The Definition of Standard ML*, '`[`E_1`, ..., `E_n`]`' is a shorthand for 'E_1`::...::`E_n`::nil`', where `::` is the infix list constructor. We have already seen how we translate infix function application, so we could iterate that to get *app-seq(val(::), tuple-seq(*$E_1^\top$ *app-seq(val(::), ... app-seq(val(::), tuple-seq(*$E_n^\top$ *list()))...)))*. The empty list `nil` is represented by the value *list()*.

Expressions with side-effects can be written using the data constructor '`ref`', which computes an updatable reference to a value, and the infix operation '`:=`', which can be used to update references. Both of them are part of the *Initial Basis of ML* [18, App. D]. It is also possible to give action semantic descriptions of these operations, but we shall omit the details here.

3.2 Patterns

The function used for mapping ML patterns to BAS parameters is named *pat2bas*. The complete definition of it can be found in App. C.1. In this subsection we will only elaborate on the subset of the mapping displayed in Table 4.

_	$\rightarrow$	*anon*
C	$\rightarrow$	C
I	$\rightarrow$	*val-or-var(*I*)*
`(`P_1`, ..., `P_n`)`	$\rightarrow$	*tuple(*$P_1^\top$ ... $P_n^\top$*)*
$I\ P$	$\rightarrow$	*app(val(*I*),* $P^\top$*)*
`[`P_1`, ..., `P_n`]`	$\rightarrow$	*app(list, tuple(*$P_1^\top$ ... $P_n^\top$*))*

Table 4: ML patterns to BAS mapping

The simplest pattern '_' is mapped to *anon*. Since the meaning of '_' is that it matches all values, one might think that we could regard it as an identifier which also matches all values; but this would generate a binding from '_' to the value, which is not the intention of this pattern.

In BAS, the sort of constants is a subsort of patterns, so a constant in the concrete syntax is just mapped to the same constant. The identifier pattern can either be a data constant bound to a value (like `true` or `nil`) or it can be an identifier matching any value. This context-dependent interpretation is represented by the construct *val-or-var(I)*.

BAS also contains a tuple pattern *tuple*($P_1 \ldots P_n$), which matches tuple values, by matching each component in the tuple value against the pattern at the same position in the tuple pattern, and joining the computed bindings. The BAS tuple pattern is the obvious target of the ML tuple pattern.

The ML pattern '$I\,P$' is mapped to *app(val(I), $P^\top$)*. The construct *app(E, P)* matches values that can be obtained by applying the function computed by E to an argument that matches P. The expression *val* was explained in a previous subsection.

List patterns (Table 4) are very similar to list expressions (Table 3) when mapped to BAS, since expressions construct values and patterns de-construct them.

In ML, the order in which subpatterns constituting a pattern are matched doesn't matter, and therefore none of the BAS constructs used in this subsection insist on sequential evaluation.

3.3 Declarations

Appendix C.3 defines the function *dec2bas*, which maps declarations to BAS. In this section we will give some illustrations of its definition. The illustrations are listed in Table 5

`val` I `=` E	$\rightarrow$	*bind-val(val(I), $E^\top$)*
`fun` IP `=` E	$\rightarrow$	*rec(bind-val(var(I), abs($P^\top$, $E^\top$)))*
D_1 `;` D_2	$\rightarrow$	*accum($D_1^\top$, $D_2^\top$)*
`local` D_1 `in` D_2 `end`	$\rightarrow$	*local($D_1^\top$, $D_2^\top$)*
`datatype` I=I_1 `of` T_1		*simult-seq(bind-val(val(I_1), new-cons)*
`| ...`	$\rightarrow$	...
`|` I_n `of` T_n		*bind-val(val(I_n), new-cons))*
`exception` I `of` T	$\rightarrow$	*bind-val(val(I), new-cons)*

Table 5: ML declarations to BAS mapping

The simple binding of a value to an identifier is a special case of the construct '`val` P `=` E', where P ranges over patterns. This is mapped to *bind-val($P^\top$, $E^\top$)* with the semantics that $E^\top$ is evaluated and then matched against $P^\top$ to create bindings.

Recursive functions in ML have the most complicated mapping to BAS that we have encountered; this is especially visible in the mapping described in App. C.3. The mapping of '`fun` IP `=` E' can be found in Table 5, and it contains some of the BAS constructs introduced previously, but also the new construct *rec(D)*, which ensures that the bindings given by D are recursive.

A sequence of declarations 'D_1`;`D_2' is directly mapped to *accum($D_1^\top$, $D_2^\top$)*, which accumulates declarations while allowing the declarations in $D_2^\top$ to override the declarations in $D_1^\top$.

Local declarations '`local` D_1 `in` D_2 `end`' can also be translated directly to a single BAS construct, namely *local($D_1^\top$, $D_2^\top$)*. The semantics of *local(D_1, D_2)* is that first the declarations generated by D_1 together with the previous declarations can be used in D_2, but the result is only the declarations generated by D_2.

With respect to datatype declarations we are only interested in the data constructors. The name of the constructor is bound to a fresh constructor (*new-cons*) using *bind-val*. The bindings are collected using *simult-seq*, which reflects that the declarations are independent and an identifier is only bound once in a datatype declaration.

Exception declaration is similar to datatype declaration in that we are only interested in the name of the exception, which is bound to a fresh constructor (*new-cons*). We don't see any reason to distinguish between exception constructors and data constructors.

4 Action Semantics for Basic Abstract Syntax

In this section we will describe the semantics of selected BAS constructs using Action Semantics (AS). The rest of the constructs used in the description of Core ML can be found in App. D. Table 6 gives an example of a BAS construct and its mapping to an action.

As we have seen in the previous section we can describe ML constructs using BAS constructs in a relatively brief and precise way. This section will show that we can also give semantics to the BAS constructs in an uncomplicated but still formal way, by using AS. Giving an AS of every ML construct directly would make the description much more complicated, because the BAS constructs allow us to decompose the ML constructs into simpler constructs, which are then described individually.

```
evaluate
  local(
    bind-val(val-or-var(x), 3),
    cond(true, val(x), 5)
  )
⇒

  ((furthermore
      give the val 3)
    then
        ((maybe
            (check not (def(the cons bound-to the token x))
          then
             bind (the token x, the val)))
      else
          ((maybe
             check (val(the cons bound-to the token x)
                                             = the val))
        then
          give no-bindings)))
scope
     ((give the val true
   then
         ((maybe
           check the boolean)
       then
           ((maybe
             give val (the cons bound-to the token x))
         else
           give the val bound-to the token x))
 else
   give the val 5)
```

Table 6: Mapping BAS to AS

4.1 Action Semantics

When giving an action semantic description of a BAS construct we shall use the Action Semantics Definition Formalism (ASDF). An ASDF module describes a single language construct and consists of different sections. The **syntax** section contains the abstract syntax of the construct. The **requires** section contains description of the data, data operations, variables etc. used to write the semantic function. The semantic function, which maps the construct to an action, is defined in the **semantics** section.

The actions in AS are written using Action Notation (AN [16, 21][4]), a notation resembling English but still strictly formal. Actions are constructed from yielders, action constants and action combinators, where yielders again consist of data, data operations and predicates.

The performance of an action might be seen as evaluation of a function from data and bindings to data, with side effects like changing storage and sending messages. The action combinators are different ways of combining functions to obtain different kinds of control and data flow in the evaluation. The evaluation can terminate in three different ways: *normally* (the performance of the enclosing action continues normally), *abruptly* (the enclosing action is skipped until the exception is handled) or *failing* (corresponding to abandoning the current alternative of a choice and trying alternative actions). AN has actions to represent evaluation of expressions, declarations, abstractions, manipulation of storage and communication between agents. The yielders can be used to inspect and create data and bindings. In the following we will introduce a considerable subset of AN, giving examples of its use.

4.2 Expressions

For every syntactic sort we have a module introducing a variable ranging over this sort, the signature of the semantic function mapping the sort to an action and other things which are common to all the modules defining the constructs belonging to this syntactic sort. Below is shown the module for *Exp*.

Module 1 *Exp*

requires

E : *Exp*

[4]The new version of AN proposed in the two referenced papers has not yet been frozen, and is a bit different from the one used in this paper, but the exact details of AN are not important here.

Datum ::= *Val*

semantics evaluate: *Exp* → *Action*

The module defines the semantic function evaluate and variables starting with *E* to range over expressions. Furthermore values in *Val* are injected into the sort *Datum* which is the sort describing items of data in AN.

The simplest expressions are constant values. The following module describes values as expressions.

Module 2 *Exp/Val*

syntax *Exp* ::= *Val*

semantics evaluate *V* = give the val *V*

Values are a subsort of expressions and have a very simple mapping to AN. The variable *V* ranging over values is declared in the module *Val*, which is automatically imported because the syntactic sort *Val* is used in this module. We use the action constant result *V*, which gives the value *V* as its result.

Notice that we use the same notation for injecting one sort into another, regardless of whether the sorts concerned are for abstract syntax or data.

The following module defines the *val(I)* construct:

Module 3 *Exp/Val-Id-Const*

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

requires *Val* ::= *Cons*

Cons ::= cons(val: *Val*)

Bindable ::= *Val*

semantics evaluate val(*I*) =
maybe give val(the cons bound-to the token *I*)
else give the val bound-to the token *I*

Constructed data belongs to the sort *Cons*. The data operation val is used to construct a value from constructed data. In the action the yielder the cons bound-to *I* is used, which looks up the constructed data bound to *I* in the current bindings. It is the data operation the cons, which ensures that *I* is bound to constructed data. The action constant give applies a yielder to the given data. If *I* was bound to something not of sort Cons the give action would terminate exceptionally and maybe and else makes sure that an alternative is tried. The alternative is to try to give the value bound to *I*, which might also terminate exceptionally, depending on the context.

The action becomes more complicated when we look at the *app-seq* construct defined in this module:

Module 4 *Exp/App-Seq*

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

requires *Val* ::= *Func*

func-no-apply : *Val*

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

In the requires section we make sure that functions and the special exception value func-no-apply are included in values. Informally, the action in the semantic function starts by evaluating *E1* and then evaluates *E2*. The action combinator and-then concatenates the results of evaluating the two subactions and the then combinator gives this result to the next action. Again we see the use of the data operation the *ds*, where *ds* identifies a data sort; in this case the func is used to ensure that *E1* evaluates to an element of *Func*, the sort of data used to represent function abstractions. The data operation #*n* selects the *n*th component of a sequence of data items. The operation action is a selector on the datatype *Func*, selecting the action to be enacted when applying a function. The action constant apply is given an action (as data) and a value, and the given value is passed to the enaction of the given action. If apply fails, the else action combinator ensures that the alternative action throw func-no-apply is performed so that the whole action terminates exceptionally. If the application doesn't fail, the result of the whole action is just the result of the application.

The result of applying a data constructor *cons* to a value *val* is the tagged value *tag(cons, val)*.

The semantics of the conditional expression is that a boolean expression is evaluated to decide which one of two expressions should be evaluated and give the result of the whole expression. The definition looks as follows.

Module 5 *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*

The first expression must evaluate to a boolean, so booleans should be included in values; this is described in the requires section. The action constant check *Y* evaluates the yielder *Y* with the given data, and if it evaluates to *true* the action terminates normally, giving the input data; otherwise it terminates exceptionally, giving no data. Combined with the maybe action combinator, which fails when the action it is combined with terminates exceptionally, we get the effect of checking whether *E1* evaluates to *true* or *false*. Connected with the now familiar else action combinator, the result is a choice between the evaluation of *E1* and that of *E2*.

Declarations local to an expression are described using the *local* construct:

Module 6 *Exp/Local*

syntax *Exp* ::= local(*Dec*, *Exp*)

semantics evaluate local(*D*, *E*) =
furthermore declare *D* scope evaluate *E*

Two new action combinators are introduced above. The prefix combinator furthermore *A* performs the action *A*, which is supposed to compute bindings; the result is the current bindings overridden by the computed bindings.[5] The

[5]The result of overriding bindings B_1 with bindings B_2 is the union of B_2 and the bindings occurring in B_1 but not in B_2.

infix combinator *A1* scope *A2* performs *A1*, which is supposed to compute bindings, and these are the bindings current when performing *A2*.

Abstractions involve two facets of AS: actions as data, and scopes of bindings.

Module 7 *Exp/Abs*

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

requires *Val* ::= *Func*

semantics evaluate abs(*P*, *E*) =
give func(closure(furthermore match *P* scope evaluate *E*))

The *abs* construct uses functions so again they are required to be included in values. When matching a parameter, bindings are generated and the action combinator furthermore makes sure that they override the current bindings. The resulting bindings become the current bindings when evaluating the expression because of the behaviour of the scope action combinator. This action is used as data when a closure is computed and then the data constructor func is applied to get a function before the result is given. We see that a function consists of an action, which is the action being applied in the description of the *app-seq* expression. The use of furthermore and scope here is similar to the way they are used in the description of the *local* construct, which seems natural since the bindings generated by the parameters have local scope.

We will skip the module defining the construct *throw*, because it doesn't introduce any new AN, and instead we will take a look at another module concerned with exceptions.

Module 8 *Exp/Catch*

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

requires *Val* ::= *Func*

semantics evaluate catch(*E1*, *E2*) =
evaluate *E1* catch
(evaluate *E2* and give the val
then apply (action(the func#1), the val#2)
else throw the val)

The *catch* construct first evaluates the expression *E1*. If the evaluation terminates exceptionally, the catch action combinator ensures that the data thrown by *E1* is given to the function to which expression *E2* evaluates. If the function cannot be applied to the result, the result is thrown again.

4.3 Statements

Although ML doesn't contain statements as such, some of its constructs correspond closely to familiar kinds of statements, and we can define their semantics by mapping them to BAS statement constructs such as the *while* construct:

Module 9 *Stm/While*

syntax *Stm* ::= while(*Exp*, *Stm*)

requires *Val* ::= *Boolean*

semantics execute while(*E*, *S*) =
unfolding (evaluate *E* then
maybe check the boolean then
execute *S* then unfold
else skip)

The iteration in the *while* construct is performed by the unfolding *A* and unfold actions. The action constant unfold performs the action *A* of the smallest enclosing occurrence of unfolding *A*.

4.4 Parameters

In ML patterns, an identifier can have two meanings: it can either be a data constructor, which matches only the same constant value; or it can be an ordinary identifier, which matches every value. The parameter construct *val-or-var* catches both meanings, by simply trying each one of them.

Module 10 *Par/Val-Or-Var*

syntax *Par* ::= val-or-var(*Ide*)

requires *Val* ::= *Cons*

Cons ::= cons(val:*Val*)

Bindable ::= *Val*

semantics match val-or-var(*I*) =
    (given val(the cons bound-to the token *I*)
      then give no-bindings)
    else
      bind(the token *I*, the val)

Constructed data belongs to the sort *Cons*, but we can construct a value from constructed data using the data operation val. The construct *val-or-var(I)* is mapped to an action that first checks whether *I* is not bound to a data constructor and then binds *I* to the given value. Otherwise it compares the given value with the constructor bound to I, using the action constant given Y, which checks that the received data is equal to the data computed by the yielder.

The following module contains the definition of the parameter construct app which matches constructed values.

Module 11 *Par/App*

syntax *Par* ::= app(*Exp*, *Par*)

requires

Val ::= *Func*

PrefixDataOp ::= invert

semantics match app(*E*, *P*) =
    give the val and
    evaluate *E* then
    maybe give invert(the func#2, the val#1)
    then match *P*

The technique here is to use the data operation invert, which takes an invertible function (such as a data constructor) and a value, and applies the inverse of the function to the value. The result of this is then matched against the parameter.

4.5 Declarations

The simplest way of binding is matching a value against a parameter which computes a set of bindings. This is described by the *bind-val* construct shown below.

Module 12 *Dec/Bind-Val*

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

semantics declare bind-val(*P*, *E*) = evaluate *E* then match *P*

The construct is mapped to an action which first evaluates the expression and then matches the parameter with the result.

More interesting is the construct *simult-seq* which describes simultaneous sequential declarations.

Module 13 *Dec/Simult-Seq*

syntax *Dec* ::= simult-seq(*Dec*+)

semantics

declare simult-seq(*D*) = declare *D*

declare simult-seq(*D* *D*+) =
  declare *D* and-then
  declare simult-seq(*D*+) then
  give disj-union

Two equations are used to define the semantics of the *simult-seq* construct. If the sequence just consists of a single declaration, it just declares it, otherwise it declares the first declaration in the sequence and then declares simultaneously the rest. Finally it computes the disjoint union of the bindings resulting from the two recursive applications of the semantic function.

The construct *rec(D)* allows recursive declarations where the bindings computed from *D* can be used in the functions and procedures declared in *D*.

Module 14 *Dec/Rec*

syntax *Dec* ::= rec(*Dec*)

semantics declare rec(*D*) = recursively declare *D*

AN contains an action combinator that does exactly this, it is called recursively.

When a sequence of declarations accumulates bindings while letting a declaration redefine the previous declarations, one uses the *accum* construct shown below.

Module 15 *Dec/Accum*

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

semantics

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

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

The interesting part here is the new action combinator before, which takes the bindings computed by the action on the left-hand side and lets the right-hand side action use them before it overrides them with the bindings computed by the right-hand side action.

5 Reusability

The foregoing sections have explained the overall organisation of a constructive action semantics of Core ML, and illustrated the various parts of it. Let us now assess the degree of reusability that we have obtained in the various parts of it.

5.1 The Syntax of Core ML

We have chosen to start from the syntax for Core ML given in *The Definition* [18, App. B], reformulated as a grammar in SDF as shown in App. A. Although *The Definition* interprets the grammar as abstract syntax in connection with specifying the semantics of ML, the grammar is also used to define the concrete syntax of ML, and involves not only (relative) priorities but also rather more nonterminal symbols than one would expect in an abstract syntax.

Starting from this grammar has both advantages and disadvantages. On the positive side, we can give semantics directly to real program texts, parsed exactly as they would be by (conforming) implementations of ML. The reformulation in SDF was not entirely trivial, but a lot less effort than it would be to develop an alternative grammar for ML from scratch. One drawback is that the grammar is somewhat larger than a typical grammar for abstract syntax would be; another is that the various parts of it cannot easily be reused in descriptions of other languages.

It is also worth noting that *complete* descriptions of languages inherently involve concrete syntax, but are seldom given in connection with semantic descriptions.

5.2 Mapping from Core ML to BAS

The complete mapping is specified in App. C, in ASF. Clearly, we need at least one rule per Core ML construct, which almost entirely accounts for the length of the specification. The individual rules are mostly very simple, mapping an ML construct either directly to a BAS construct, or to a simple combination of BAS constructs. We found the expansion of 'fun' declaration given in *The Definition* [18, App. A] somewhat clumsy, so we use a simpler translation, totally avoiding the need for creating 'fresh' variable identifiers. The basic idea is illustrated in Table 7.

$$\begin{array}{lll} \texttt{fun} & I\ P_{11} \ldots P_{1m} & = & E_1 \\ \quad | & \ldots & & \\ \quad | & I\ P_{n1} \ldots P_{nm} & = & E_n \end{array}$$

$\Rightarrow$

$$\begin{array}{llll} \texttt{val rec}\ I = \mathit{curry}_m & (\texttt{fn} & (P_{11},\ \ldots,\ P_{1m}) & \texttt{=>} & E_1 \\ & \quad | & \ldots & & \\ & \quad | & (P_{n1},\ \ldots,\ P_{nm}) & \texttt{=>} & E_n) \end{array}$$

where

$$\mathit{curry}_m = \texttt{fn}\ f\ \texttt{=>}\ \texttt{fn}\ v_1\ \texttt{=>}\ \ldots \texttt{=>}\ \texttt{fn}\ v_m\ \texttt{=>}\ f(v_1, \ldots, v_m)$$

Table 7: Expansion of 'fun' declarations

Although some of the rules look as if they could be reusable, it appears to be more trouble than it is worth to make a separate module for each Core

ML construct and the rule translating it to BAS.

It might be preferable to integrate the specification of the translation from Core ML to BAS with that of the concrete syntax of Core ML, as can be done using logic grammars in Prolog, and (less elegantly) in yacc grammars. The simple translation from mixfix to prefix constructors available in SDF grammars is clearly inadequate for our purposes, but we do not need the full generality provided by ASF.

5.3 Action Semantics of BAS

The basis for our constructive action semantics of Core ML is the collection of modules defining the action semantics of the individual BAS constructs. Since each BAS construct has been designed to represent a single programming feature, its action semantics is often significantly simpler than that of typical Core ML constructs. Almost all the BAS constructs are highly reusable, and not biased or specific to representation of Core ML constructs. In particular, we are able to reuse constructs concerning *statements* in connection with describing ML's sequencing and while-expressions (by exploiting constructs for obtaining statements from expressions and vice versa).

The main exception concerns nested parameters, used to represent ML's patterns: other languages will most likely involve tuples only of variable identifiers, rather than the tuples of arbitrary parameters provided here. However, inspection of the module concerned indicates that little would be gained by specialising it: the recursive call of the semantic function 'match' on a sub-parameter, together with the separate definition of 'match' on a single identifier, are just as simple (if not simpler) than combining them both in the same equation.

One construct that has been added to BAS specifically in connection with ML is the parameter 'val-or-var(I)'. An occurrence of an identifier as a parameter of a function abstraction is interpreted as a constant if the identifier is itself a data constructor (such as '`nil`'), otherwise it is interpreted as a variable — even if it is already bound as a variable to the value of a data constructor. The (context-free) concrete syntax gives no hint about which interpretation is intended, so we are forced to map the identifier to a construct which admits both interpretations. We are not aware of other languages that would require use of this BAS construct. The need to extend BAS with a construct to be used only in connection with one language (or family of related languages) indicates that the language concerned has an unusual feature; whether that feature represents an unusually clever bit of language design, or an atypically poor one, is left open.

It should be stressed that BAS is at an early stage of development, and that the notation used for sorts and constructors may not become stable until further major case studies have been completed (e.g., significant sub-languages of C and Java, and the extension of the present case study to ML modules and to Concurrent ML). However, our translation from Core ML to BAS does not appear to be particularly sensitive to minor adjustments in the intended interpretation of BAS constructs. Changes to the spelling of symbols would of course require global editing, but that can be automated. More work might be required in connection with the introduction of subsorts or supersorts of the existing sorts of constructs. For instance, a potential refinement of BAS would be to take account of whether the execution of a construct might 'fail' or not (where failure is always to lead to an alternative, and ultimately to an infallible alternative). This would allow the description of the '*alt-seq*' construct to be simplified, but it might also require changes to some of the other rules in the translation.

6 Environment

The ASF+SDF specification of Core ML found in Apps. A, B, and C has been developed using the ASF+SDF Meta-Environment (ME) [5, 6]. Among some of the key features of the ME are:

- visualisation of the import relation between the open modules as a graph;
- visualisation of the hierarchical structure of the module names as an expandable tree;
- syntax directed editing of ASF+SDF modules and terms;
- parsing and rewriting terms over modules;
- the ability to export parse tables and equation files; and
- representing the syntax and semantics of the specified language for use with command line tools.

In Sect. 4 we have seen examples of BAS constructs defined using ASDF, our Action Semantics Definition Formalism. For working with the ASDF modules we used the Action Environment (AE) [1], a new extension of the ME, which was developed by Iversen.

The extensibility of the ME allowed us to add an extra layer that supports ASDF modules on top of the ME. This layer provides a mapping from ASDF modules to ASF+SDF modules. By using this approach, it was easy for us to implement most of the functionality of the ME (including the features mentioned in the list above). For the user, the only visible difference between the two environments is the meta-notation used to define the language modules, and minor changes to the menus.

The ME is a set of tools connected using the ToolBus. Connecting a tool to the ToolBus is done using ToolBus scripts, and the ME consists of approximately 6800 lines of ToolBus scripts. The ASDF layer in the AE consists of approximately 1400 lines of ToolBus scripts, an extension of about 20 % measured in lines of ToolBus script. But the ASDF layer also consists of an ASDF parse-table and various tools.

A screen dump of the AE is shown in Fig. 1. Notice the graph illustrating the import relations between the modules on the right side, and the expandable tree of hierarchical module names on the left side.

When giving a semantic description of a language, the user would typically start by deciding which language constructs are present in the core of the language, and which BAS modules can be used to describe these constructs. The designer then creates a new module in the AE that imports all of these modules from the library. Now he can work with terms composed of the BAS constructs by parsing them and mapping them into actions, by opening a term editor over the new module. The mapping is described in the semantic sections of the imported modules. In the near future he will also be able to evaluate the terms when we integrate an existing action evaluator [22] into the AE. Incrementally the language designer will add more constructs to the language, and perhaps even need to design new BAS constructs. He does so by creating a new module in the AE and use the syntax directed editor to write the new BAS construct and immediately have the ASDF syntax checked. Using the integrated type checker, he can also check that the semantic function has the expected type (the type checker is very liberal but gives the designer valuable hints about the validity of the semantic function). As the language grows the graph over the imported modules, and the tree containing the hierarchically structured module names, helps the language

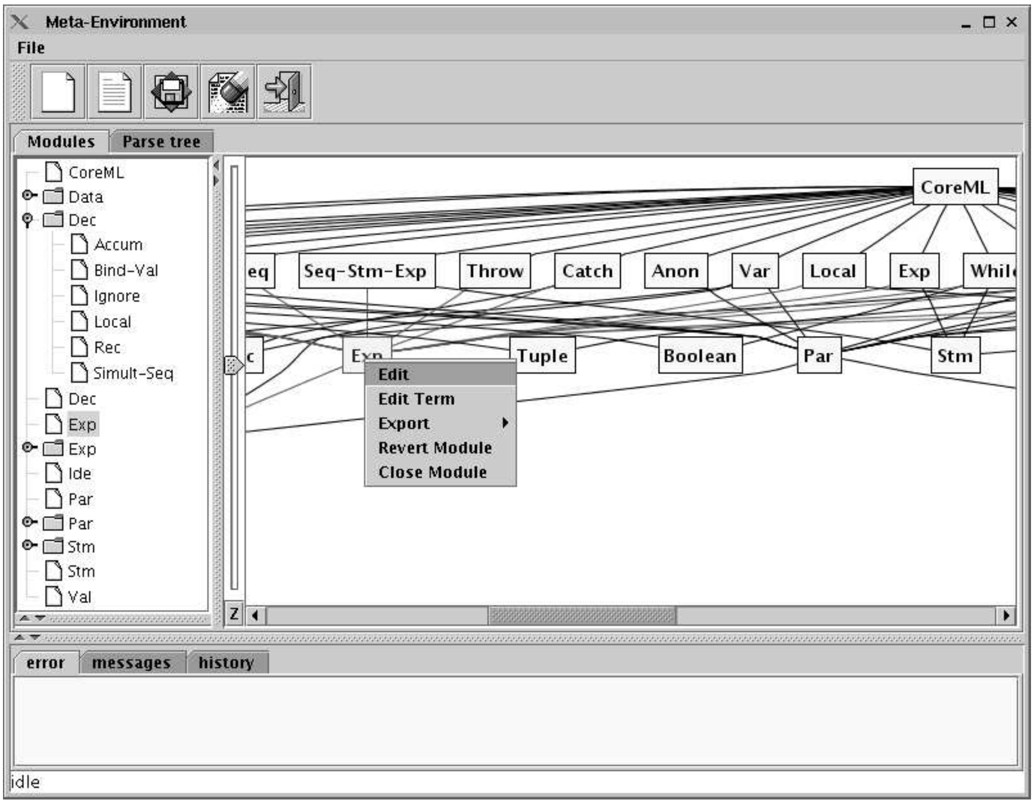

Figure 1: The Action Environment

designer to retain an overview of the description.

Combining the ME and the AE gives us tool support for mapping the concrete syntax of a language (for instance ML) to actions. Future plans involves improving the AE such that one can also work with ASF+SDF modules in it, thereby providing support for both the description of the concrete syntax and the semantics in the environment. We also plan to integrate different tools into the AE. An action compiler will turn the environment into a compiler generator. All in all, the AE combined with other tools will surely provide a particularly useful environment for developing semantic descriptions and documenting the design of programming languages.

7 Related Work

Watt [23] reported on a previous case study concerning the use of Action Semantics to describe ML, covering both the static and dynamic semantics of Core ML, and the dynamic semantics of ML modules. He also compared his description with *The Definition of Standard ML* [18]. One of the contributions of our present case study is to show how part of his description might look when refactored in the constructive style.

We have not attempted to give a reformulation of Watt's static semantics of Core ML in our constructive style. This is partly because the use of action semantics for specifying static semantics is unorthodox, and not well-known. In general, we would expect to be able to use the same expansion to BAS for both the static and the dynamic semantics, except that types have to be retained in the former, which necessitates a few extra BAS constructs. Of course, the action semantics of most BAS constructs is quite different for their static and dynamic semantics; but their overall organisation is identical.

To extend our dynamic action semantics of Core ML to describe also the semantics of ML modules would require augmenting BAS with constructs that represent the visibility of bindings in signature and structure declarations, as well as sharing relationships. This is left as an interesting topic for future work, since our aim here is not to cover a full-scale language in full detail, but rather to illustrate our basic approach on a sizable collection of realistic constructs.

In Table 8 we have illustrated the parts of Watts description that describe the semantics of the 'if-then-else' expression and compared it to the *Exp/Cond* BAS module. The main difference is the syntax part of the descriptions and the overall structure of the descriptions. It is clear that Watts description does not have the same degree of modularity as ours.

Grammar

...

Expression = ...
                | "if" *Expression* "then" *Expression* "else" *Expression*
                | ...

...

Semantic functions

...

```
evaluate ("if" E1 : Expression "then" E2 : Expression "else" E3 : Expression) =
     evaluate E1 then
     (  (check (the given value is the boolean of true)
         then evaluate E2)
      or
        (check (the given value is the boolean of false)
         then evaluate E3)
     )
```

...

Semantic entities

...

value = ... | boolean | ...

...

Module 16 *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
```

Table 8: Watts and our description of the conditional expression

It is difficult to compare the size of Watts ML description [23] and our description due to the structural differences. Watts description has a direct mapping from ML syntax to AS, whereas ours contains both a mapping from ML syntax to BAS and from BAS to AS. This increases the size of our description. On the other hand reusing the BAS constructs many times reduces the size of our description.

Doh and Mosses [21] introduced the main ideas of constructive action semantics, and proposed changing the modular structure of action semantic descriptions accordingly. They gave illustrations of descriptions of familiar individual constructs, and showed how some idealised programming languages could be composed by importing the modules for the required constructs; here, we illustrate the composition of a real language, Core ML. The modules that they gave were written directly in ASF+SDF [19, 20]; in contrast, we provide ASDF an Action Semantics Definition Formalism, designed specifically for use in giving action semantic descriptions of individual constructs, avoiding the many tedious keywords, tiresome definition of variables and lists of 'obvious' imports that were required when using ASF+SDF, and we have implemented our meta-notation in the Action Environment, built on the ASF+SDF Meta-Environment [5, 6]. The modules in Table 9 shows ASDF and ASF+SDF modules used to specify the same construct.

In both the studies cited above, abstract syntax was deliberately very close to concrete syntax, using keywords and symbols from programs in the described language to distinguish between abstract constructs. Here, we propose a neutral Basic Abstract Syntax, BAS, and specify a mapping from concrete syntax to BAS, in the interests of increased reusability when describing languages having significantly different concrete syntax for the same abstract constructs.

8 Conclusion

In this paper, we have presented a case study of the use of Constructive Action Semantics. The case study demonstrates that an action semantics for a real language, Core ML, can be based on reusable action semantic descriptions of individual language features. The language features were taken from BAS, our Basic Abstract Syntax, but our approach is quite general, and does not depend on the exact choice of BAS constructs, nor on the details of their action semantics.

We have used SDF to specify the syntax of Core ML and BAS, and ASF to specify the translation from Core ML to BAS. We have introduced a new

Type of module	ASDF	ASF+SDF

Semantic function	**module** `M`
	requires
	  `V : S1`
	  `S2 ::= S3`
	semantics
	  `f: S1 -> Action`
		module `M`
		imports `S1 S2 S3`
		exports
		  sorts `S2`
		  context-free syntax
		    `S3      -> S2`
		    `f S1    -> Action`
		  variables
		    `"V"[1-9]?       -> S1`
		    `"V"[1-9]?"*"    -> S1*`

Semantic equations	**module** `M`
	syntax
	  `S1 ::= c(S2, ..., Sk)`
	requires
	  `Sm ::= Sn`
	semantics
	  `[] f c(V1, ..., Vk) = A`
		module `M`
		imports `S1 S2 ... Sk Sm Sn`
		exports
		  context-free syntax
		    `c(S2, ..., Sk)    -> S1`
		    `Sn       -> Sm`
		equations
		  `[] f c(V1, ..., Vk) = A`

Table 9: Comparing ASDF with ASF+SDF modules

formalism, ASDF (Action Semantics Definition Formalism), to eliminate the many tedious keywords and imports that are needed when using ASF+SDF directly for the specification of the action semantics of the individual BAS constructs.

Our long-term goals are to provide:

- a stable (but open-ended) library of reusable action semantic descriptions of BAS constructs, supporting constructive action semantics of all major programming languages, and
- an enhanced Action Environment, supporting realistic compiler generation from constructive action semantic descriptions, as well as the development, checking, and prototyping of the latter.

In the short-term, we will develop further case studies, extending and refining BAS as required.

Much work will be needed to accomplish the above goals. Readers interested in particular topics are invited to contact the authors regarding possibilities for cooperation.

References

[1] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. In *LDTA 2004*, 2004.

[2] The SML/NJ Fellowship. Standard ML. `http://www.smlnj.org/sml.html`.

[3] Jan Heering and Paul Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000.

[4] David A. Watt. Why don't programming language designers use formal methods? In R. Barros, editor, *Anais XXIII Seminário Integrado de Software e Hardware*, pages 1–16. Universidade Federál de Pernambuco, Recife, Brazil, 1996.

[5] 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 R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001, Genova, Italy, Proceedings*, LNCS Vol. 2027, pages 365–370. Springer, 2001.

[6] Mark G. J. van den Brand and Paul Klint. ASF+SDF *Meta-Environment User Manual*, 2003. `http://www.cwi.nl/projects/MetaEnv/meta/doc/manual.ps.gz`.

[7] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[8] Peter D. Mosses and David A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland, 1987.

[9] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

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

[11] Peter D. Mosses, editor. *AS'94, 1st Intl. Workshop on Action Semantics, Edinburgh, Proceedings*, BRICS NS-94-1. Dept. of Computer Science, Univ. of Aarhus, 1994.

[12] Peter D. Mosses. A tutorial on action semantics. Technical Report BRICS NS-96-14, Dept. of Computer Science, Univ. of Aarhus, 1996. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996).

[13] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland, Proceedings*, LNCS Vol. 1113, pages 37–61. Springer, 1996.

[14] Peter D. Mosses and David A. Watt, editors. *AS'99, 2nd International Workshop on Action Semantics, Amsterdam, The Netherlands, Proceedings*, BRICS NS-99-3. Dept. of Computer Science, Univ. of Aarhus, 1999.

[15] Peter D. Mosses and Hermano Moura, editors. *AS 2000, 3rd International Workshop on Action Semantics, Recife, Brazil, Proceedings*, BRICS NS-00-6. Dept. of Computer Science, Univ. of Aarhus, 2000.

[16] Søren B. Lassen, Peter D. Mosses, and David A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In Mosses and Moura [15], pages 19–36.

[17] Peter D. Mosses, editor. *AS 2002, 4th International Workshop on Action Semantics, Copenhagen, Denmark, Proceedings*, BRICS NS-02-8. Dept. of Computer Science, Univ. of Aarhus, 2002.

[18] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[19] Jan A. Bergstra, Jan Heering, and Paul Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. Addison-Wesley, 1989.

[20] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing Vol. 5. World Scientific, 1996.

[21] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

[22] Tijs van der Storm. An-2 tools. In *AS 2002*, pages 23–42. BRICS, December 2002.

[23] David A. Watt. The static and dynamic semantics of SML. In Mosses and Watt [14].

A The syntax of core ML

SDF, the Syntax Definition Formalism [19, 20] is developed at CWI[6] and it allows all context-free grammars. An SDF specification consists of a set of modules, each defining parts of the full syntax. A module can use syntactic sorts (nonterminals) defined in other modules by importing these modules. In SDF context-free and lexical syntax is mixed. Table 10 shows an example of a small module named *Expressions*, which imports the modules *Integers* and *Identifiers* and defines the syntactic sort *Exp*. Notice that SDF uses an uncommon notation in production rules: instead of the usual ::= operation it uses → with the sort being defined on the right-hand side. The defining part of a production rule can contain lexical syntax (`"+"`), sorts (*Exp*) and regular expressions ({*Exp* `","` }`*`, a comma separated sequence of *Exp*'s). To resolve ambiguities, SDF lets the user define associativity after production rules ({`left`}) and priorities in the "context-free priorities" section (examples of this can be found in App. A.1).

module *Expressions*

imports *Integers* *Identifiers*

exports

sorts *Exp*

context-free syntax

Identifier → *Expression*
Integer → *Expression*
Expression `"+"` *Expression* → *Expression* {`left`}
`"("` {*Expression* `","` }`*` `")"` → *Expression*

Table 10: SDF example

For readability we have used longer names in this appendix for the syntactic sorts introduced in Sect. 2. We use *ValueId* instead of *IDE*, *Constant* instead of *CON*, *Expression* instead of *EXP*, *Pattern* instead of *PAT*, *Declaration* instead of *DEC* and *Type* instead of *TYP*.

[6] `http://www.cwi.nl`

A.1 Expressions

module *Expressions*

imports

Constants
Identifiers
Patterns
Declarations

exports

sorts

Expression AtomicExp InfixExp ApplicationExp ExpRow
SingleExpRow Match MatchRule

context-free syntax

%% Atomic Expressions

Constant	→	*AtomicExp*
“op”? *LongValueId*	→	*AtomicExp*
“{” *ExpRow*? “}”	→	*AtomicExp*
“#” *Label*	→	*AtomicExp*
“(” “)”	→	*AtomicExp*
“(” *Expression* “)”	→	*AtomicExp*

“(” *Expression* “,” { *Expression* “,” }+ “)”	→	*AtomicExp*
“[” { *Expression* “,” }* “]”	→	*AtomicExp*
“(” *Expression* “;” { *Expression* “;” }+ “)”	→	*AtomicExp*
“let” *Declaration* “in” { *Expression* “;” }+ “end”	→	*AtomicExp*

Label “=” *Expression*	→	*SingleExpRow*
{ *SingleExpRow* “,” }+	→	*ExpRow*

%% Application Expression

AtomicExp	→	*ApplicationExp*
ApplicationExp AtomicExp	→	*ApplicationExp*

%% Infix Expression

ApplicationExp	→	*InfixExp*

%% Expressions

InfixExp	→	*Expression*
Expression ":" *Type*	→	*Expression*
Expression "andalso" *Expression*	→	*Expression* {left}
Expression "orelse" *Expression*	→	*Expression* {left}
Expression "handle" *Match*	→	*Expression*
"raise" *Expression*	→	*Expression*

"if" *Expression* "then" *Expression* "else" *Expression*	→	*Expression*
"while" *Expression* "do" *Expression*	→	*Expression*
"case" *Expression* "of" *Match*	→	*Expression*
"fn" *Match*	→	*Expression*

%% Match

{ *MatchRule* "|" }+	→	*Match*
Pattern "=>" *Expression*	→	*MatchRule*

context-free priorities

{*InfixExp*	→	*Expression*
Expression ":" *Type*	→	*Expression*} >
{ *Expression* "andalso" *Expression*	→	*Expression*} >
{ *Expression* "orelse" *Expression*	→	*Expression*} >
{ *Expression* "handle" *Match*	→	*Expression*} >
{"raise" *Expression*	→	*Expression*
"if" *Expression* "then" *Expression* "else" *Expression*	→	*Expression*
"while" *Expression* "do" *Expression*	→	*Expression*
"case" *Expression* "of" *Match*	→	*Expression*
"fn" *Match*	→	*Expression*}

hiddens sorts

{ *Expression* "," }+ { *Expression* ";" }+

A.2 Patterns

module *Patterns*

imports

Types
Identifiers
Constants

exports

sorts

Pattern AtomicPattern SinglePatternRow PatternRow

context-free syntax

%% Atomic pattern

"_"	→	*AtomicPattern*
Constant	→	*AtomicPattern*
"op"? *LongValueId*	→	*AtomicPattern*
"{" *PatternRow*? "}"	→	*AtomicPattern*
"(" ")"	→	*AtomicPattern*

"(" *Pattern* "," { *Pattern* "," }+ ")"	→	*AtomicPattern*
"[" { *Pattern* "," }* "]"	→	*AtomicPattern*
"(" *Pattern* ")"	→	*AtomicPattern*

%% Pattern row

"..."	→	*SinglePatternRow*
Label "=" *Pattern*	→	*SinglePatternRow*
ValueId (":" *Type*)? ("as" *Pattern*)?	→	*SinglePatternRow*
{ *SinglePatternRow* "," }+	→	*PatternRow*

%% Pattern

AtomicPattern	→	*Pattern*
"op"? *LongValueId AtomicPattern*	→	*Pattern*
Pattern ValueId Pattern	→	*Pattern*
Pattern ":" *Type*	→	*Pattern*
"op"? *ValueId* (":" *Type*)? "as" *Pattern*	→	*Pattern*

A.3 Declarations

module *Declarations*

imports

Expressions
Patterns

exports

sorts

Declaration SingleExcBinding ExcBinding SingleConsBinding ConsBinding SingleDataBinding DataBinding SingleTypeBinding TypeBinding SingleFunValueBindingBar SingleFunValueBinding FunValueBinding SingleValueBinding ValueBinding TypeVarSequence

context-free syntax

%% Declarations

"val" *TypeVarSequence ValueBinding*	→	*Declaration*
"fun" *TypeVarSequence FunValueBinding*	→	*Declaration*
"type" *TypeBinding*	→	*Declaration*

"datatype" *DataBinding* ("withtype" *TypeBinding*)?	→	*Declaration*
"datatype" *TypeConstructor* "=" "datatype" *LongTypeConstructor*	→	*Declaration*
"abstype" *DataBinding* ("withtype" *TypeBinding*)? "with" *Declaration* "end"	→	*Declaration*

"exception" *ExcBinding*	→	*Declaration*
"local" *Declaration* "in" *Declaration* "end"	→	*Declaration*
"open" *LongStringId**	→	*Declaration*
Declaration ";"? *Declaration*	→	*Declaration* {left}
"infix" *Digit*? *ValueId*+	→	*Declaration*
"infixr" *Digit*? *ValueId*+	→	*Declaration*
"nonfix" *ValueId*+	→	*Declaration*

%% Value Binding

Pattern "=" *Expression*	→	*SingleValueBinding*
SingleValueBinding	→	*ValueBinding*
SingleValueBinding "and" *ValueBinding*	→	*ValueBinding*
"rec" *ValueBinding*	→	*ValueBinding*

%% Function Value Binding

"op"? *ValueId AtomicPattern+* (":" *Type*)? "=" *Expression*	→	*SingleFunValueBinding*
SingleFunValueBinding	→	*SingleFunValueBindingBar*
SingleFunValueBinding "|" *SingleFunValueBindingBar*	→	*SingleFunValueBindingBar*
SingleFunValueBindingBar	→	*FunValueBinding*
SingleFunValueBindingBar "and" *FunValueBinding*	→	*FunValueBinding*

%% Type Binding

TypeVarSequence TypeConstructor "=" *Type*	→	*SingleTypeBinding*
{ *SingleTypeBinding* "and" }+	→	*TypeBinding*

%% Data Binding

TypeVarSequence TypeConstructor "=" *ConsBinding*	→	*SingleDataBinding*
SingleDataBinding	→	*DataBinding*
SingleDataBinding "and" *DataBinding*	→	*DataBinding*

%% Constructor Binding

"op"? *ValueId* ("of" *Type*)?	→	*SingleConsBinding*
SingleConsBinding	→	*ConsBinding*
SingleConsBinding "|" *ConsBinding*	→	*ConsBinding*

%% Exception Binding

"op"? *ValueId* ("of" *Type*)?	→	*SingleExcBinding*
"op"? *ValueId* "=" "op"? *LongValueId*	→	*SingleExcBinding*
SingleExcBinding	→	*ExcBinding*
SingleExcBinding "and" *ExcBinding*	→	*ExcBinding*

%% Type variable sequence

TypeVariable?	→	*TypeVarSequence*
"(" { *TypeVariable* "," }+ ")"	→	*TypeVarSequence*

A.4 Types

module *Types*

imports

Identifiers

exports

sorts

Type TypeRow SingleTypeRow TypeSequence

context-free syntax

%% Types

TypeVariable	→	*Type*
"{" *TypeRow*? "}"	→	*Type*
TypeSequence LongTypeConstructor	→	*Type* {avoid}
Type "*" *Type*	→	*Type* {left}
Type "->" *Type*	→	*Type* {left}
"(" *Type* ")"	→	*Type*

%% Type row

Label ":" *Type*	→	*SingleTypeRow*
{ *SingleTypeRow* "," }+	→	*TypeRow*

%% Type sequence

Type?	→	*TypeSequence*
"(" { *Type* "," }+ ")"	→	*TypeSequence*

context-free priorities

%% Priorities

{ *TypeSequence LongTypeConstructor*	→	*Type* } >
{ *Type* "*" *Type*	→	*Type* } >
{ *Type* "->" *Type*	→	*Type* }

B SDF specification of BAS constructs

module *BAS*

imports
Layout
Identifiers
Constants

exports

sorts

Val Exp Stm Dec Pat

context-free syntax

%% Values

Con	$\rightarrow$	*Val*
label	$\rightarrow$	*Val*
list()	$\rightarrow$	*Val*
list	$\rightarrow$	*Val*
false	$\rightarrow$	*Val*
true	$\rightarrow$	*Val*
null-val	$\rightarrow$	*Val*
new-cons	$\rightarrow$	*Val*

%% Expressions

Val	$\rightarrow$	*Exp*
val(*Ide*)	$\rightarrow$	*Exp*
app-seq(*Exp*, *Exp*)	$\rightarrow$	*Exp*
tuple-seq(*Exp**)	$\rightarrow$	*Exp*
cond(*Exp*, *Exp*, *Exp*)	$\rightarrow$	*Exp*
abs(*Par*, *Exp*)	$\rightarrow$	*Exp*
alt-seq(*Exp**)	$\rightarrow$	*Exp*
seq(*Stm*, *Exp*)	$\rightarrow$	*Exp*
throw(*Exp*)	$\rightarrow$	*Exp*
catch(*Exp*, *Exp*)	$\rightarrow$	*Exp*
local(*Dec*, *Exp*)	$\rightarrow$	*Exp*

%% Statements

stm(*Exp*)	$\rightarrow$	*Stm*
while(*Exp*, *Stm*)	$\rightarrow$	*Stm*

%% Parameters

Val	$\rightarrow$	Par
anon	$\rightarrow$	Par
val-or-var(Ide)	$\rightarrow$	Par
var(Ide)	$\rightarrow$	Par
app(Exp, Par)	$\rightarrow$	Par
tuple$(Par*)$	$\rightarrow$	Par
simult$(Par*)$	$\rightarrow$	Par

%% Declarations

bind-val(Par, Exp)	$\rightarrow$	Dec
simult-seq$(Dec*)$	$\rightarrow$	Dec
rec(Dec)	$\rightarrow$	Dec
local(Dec, Dec)	$\rightarrow$	Dec
accum$(Dec*)$	$\rightarrow$	Dec
"*ignore*"	$\rightarrow$	Dec

C Mapping ML to BAS

ASF [19, 20] is essentially rewrite rules. The rules are defined using conditional equations as illustrated here

$$\texttt{[if-true]}\quad \frac{\mathit{eval}(\mathit{EX1}) \texttt{ = true}}{\mathit{eval}(\texttt{if } \mathit{EX1} \texttt{ then } \mathit{EX2} \texttt{ else } \mathit{EX3}) \texttt{ = } \mathit{eval}(\mathit{EX2})}$$

In the example we describe a function *eval* that evaluates expressions, in this case the conditional expression. The equation starts with a tag `[if-true]` naming the equation. If the tag starts with default it means that the ASF evaluator should try all non-default rules before trying this one. Above the line we have a condition, so in the case where *eval*(*EX1*) can be rewritten to `true` we can rewrite the conditional to the evaluation of the expression in the left branch. Notice that the equations can contain variables (in this case they are capitalised) ranging over syntax trees (in this case Expressions). Sometimes one also use the keyword **when** to separate the condition from the rule in conditional equations.

C.1 Expressions

In the mapping of expressions to BAS some auxiliary functions are used. *getlabels* and *getexps* are used to construct a tuple of labels and a tuple of expressions from a ML record. *label* is a data operation used to construct record values, it is applied to a tuple of labels and result is then applied to a tuple of expressions. When mapping the operation '`#`' an identifier not included in the set of ML identifiers but included in the set of BAS identifiers is needed, *special* provides this identifier. The function *expcast* is used when an expression is injected into some syntactic sort, for instance the singleton comma separated expression list.

The following list shows the signatures of the functions used in the mapping to BAS.

exp2bas	:	*Expression* → *Exp*
exp2bas	:	{ *Expression* "," }+ → *Exp**
explist2bas	:	{ *Expression* "," }+ → *Exp*
expcast	:	*Expression* → *Exp*
exp2bas	:	{ *Expression* ";" }+ → *Exp*
getlabels	:	{ *SingleExpRow* "," }+ → *Exp**
getexps	:	{ *SingleExpRow* "," }+ → *Exp**
match2bas	:	*Match* → *Exp*

The following list shows the variables used in the mapping to BAS and the syntactic sorts they range over. A number can be appended to a variable to distinguish between several occurrences of the same syntactic sort in a construct.

C	:	*Constant*	*op?*	:	"op"?
I	:	*LongValueId*	AE	:	*AtomicExp*
EX	:	*Expression*	ER	:	{ *SingleExpRow* "," }+
SER	:	*SingleExpRow*	EC	:	{ *Expression* "," }+
ES	:	{ *Expression* ";" }+	L	:	*Label*
D	:	*Declaration*	EF	:	*ApplicationExp*
T	:	*Type*	M	:	*Match*
MR	:	*MatchRule*	MP	:	{ *MatchRule* "|" }+
P	:	*Pattern*			

equations

[constant-1] *exp2bas*(C) = C

[value-id-1] *exp2bas*(*op?* I) = *val*(I)

[record-1] *exp2bas*({ }) = *app-seq*(*app-seq*(*label*, *null-val*), *null-val*)

[record-2] *exp2bas*({ ER }) =
app-seq(*app-seq*(*label*, *tuple-seq*(*getlabels*(ER))), *tuple-seq*(*getexps*(ER)))

[get-labels-1] *getlabels*(L = EX) = *val*(L)

[get-labels-2] *getlabels*(SER, ER) = *getlabels*(SER) *getlabels*(ER)

[get-exps-1] *getexps*(L = EX) = *exp2bas*(EX)

[get-exps-2] *getexps*(SER, ER) = *getexps*(SER) *getexps*(ER)

[klaf-label-1] *exp2bas*(# L) =
abs(*pat2bas*({L = *newid*, ...}), *val*(*newid*))

[tuple-1] *exp2bas*((EX, EC)) = *tuple-seq*(*exp2bas*(EX, EC))

[tuple-2] *exp2bas*(()) = *null-val*

[tuple-3] *exp2bas*(EX, EC) = *exp2bas*(EX) *exp2bas*(EC)

[tuple-4] *exp2bas*(EC) = *expcast*(EX) **when** EX = EC

[tuple-5] $expcast$(EX) = $exp2bas$(EX)

[bracket-1] $exp2bas$((EX)) = $exp2bas$(EX)

[list-1] $exp2bas$([]) = *list*()

[list-2] $exp2bas$([EC]) = *app-seq*(*list*, *tuple-seq*($exp2bas$(EC)))

[seq-1] $exp2bas$((EX ; ES)) = *seq*(*stm*($exp2bas$(EX)), $exp2bas$(ES))

[seq-2] $exp2bas$(EX ; ES) = *seq*(*stm*($exp2bas$(EX)), $exp2bas$(ES))

[seq-3] $exp2bas$(ES) = $expcast$(EX) **when** ES = EX

[let-1] $exp2bas$(let D in ES end) = *local*($dec2bas$(D), $exp2bas$(ES))

[app-seq-1] $exp2bas$(EF AE) = *app-seq*($exp2bas$(EF), $exp2bas$(AE))

[type-1] $exp2bas$(EX : T) = $exp2bas$(EX)

[andalso-1] $exp2bas$(EX1 andalso EX2) =
cond($exp2bas$(EX1), $exp2bas$(EX2), *false*)

[orelse-1] $exp2bas$(EX1 orelse EX2) =
cond($exp2bas$(EX1), *true*, $exp2bas$(EX2))

[handle-1] $exp2bas$(EX handle M) = *catch*($exp2bas$(EX), $match2bas$(M))

[raise-1] $exp2bas$(raise EX) = *throw*($exp2bas$(EX))

[if-1] $exp2bas$(if EX1 then EX2 else EX3) =
cond($exp2bas$(EX1), $exp2bas$(EX2), $exp2bas$(EX3))

[while-1] $exp2bas$(while EX1 do EX2) =
seq(*while*($exp2bas$(EX1), *stm*($exp2bas$(EX2))), *null-val*)

[case-1] $exp2bas$(case EX of M) = *app-seq*($match2bas$(M), $exp2bas$(EX))

[fn-1] $exp2bas$(fn M) = $match2bas$(M)

[match-1] $match2bas$(P => EX) = *abs*($pat2bas$(P), $exp2bas$(EX))

[match-2] $match2bas$(MR | MP) = *alt-seq*($match2bas$(MR) $match2bas$(MP))

C.2 Patterns

As in the translation of record expressions we also use auxiliary functions when mapping ML record patterns.

The following list shows the signatures of the functions used in the mapping to BAS.

pat2bas :	*Pattern* $\rightarrow$ *Par*
getlabels:	{ *SinglePatternRow* "," }+ $\rightarrow$ *Exp**
getpatts :	{ *SinglePatternRow* "," }+ $\rightarrow$ *Par**
pat2bas :	{ *Pattern* "," }+ $\rightarrow$ *Par**
pat2bas :	*AtomicPattern+* $\rightarrow$ *Par**
patcast :	*Pattern* $\rightarrow$ *Par*

The following list shows the variables used in the mapping to BAS and the syntactic sorts they range over.

PA	:	*Pattern*	*C*	:	*Constant*
LI	:	*LongValueId*	*I*	:	*ValueId*
PR	:	{ *SinglePatternRow* "," }+	*SPR*	:	*SinglePatternRow*
L	:	*Label*	*T*	:	*Type*
PC	:	{ *Pattern* "," }+	"op?"	:	"op"?
AP	:	*AtomicPattern*	*APS*	:	*AtomicPattern+*
CTY?	:	(":" Type)?			

equations

[anon-1] *pat2bas*(_) = *anon*

[constant-1] *pat2bas*(*C*) = *C*

[id-1] *pat2bas*(*op? LI*) = *val-or-var*(*LI*)

[record-1] *pat2bas*({ }) = *app*(*app-seq*(*label*, *null-val*), *null-val*)

[record-2] *pat2bas*({ *PR* }) =
app(*app-seq*(*label*, *tuple-seq*(*getlabels*(*PR*))), *tuple*(*getpatts*(*PR*)))

[get-labels-1] *getlabels*(*L* = *PA*) = *val*(*L*)

[get-labels-2] *getlabels*(...) = *val*(*label*("."".""."))

[get-labels-3a] *getlabels*(*I CTY?* as *PA*) = *val*(*I*)

[get-labels-3b] *getlabels*(*I CTY?*) = *val*(*I*) [get-labels-4] *getlabels*(*SPR*, *PR*) = *getlabels*(*SPR*) *getlabels*(*PR*)

[get-patts-1] *getpatts*(*L* = *PA*) = *pat2bas*(*PA*)

[get-patts-2] *getpatts*(...) = *anon*

[get-patts-3a] *getpatts*(*I* *CTY?* as *PA*) = *pat2bas*(*I* *CTY?* as *PA*)

[get-patts-3b] *getpatts*(*I* *CTY?*) = *val-or-var*(*I*)

[get-patts-4] *getpatts*(*SPR*, *PR*) = *getpatts*(*SPR*) *getpatts*(*PR*)

[tuple-1] *pat2bas*(()) = *null-val*

[tuple-2] *pat2bas*((*PA*, *PC*)) =
tuple(*pat2bas*(*PA*, *PC*))

[tuple-3] *pat2bas*(*PA*, *PC*) = *pat2bas*(*PA*) *pat2bas*(*PC*)

[tuple-4] *pat2bas*(*PC*) = *patcast*(*PA*) **when** *PC* = *PA*

[tuple-5] *patcast*(*PA*) = *pat2bas*(*PA*)

[list-1] *patlist2bas*() = *list*()

[list-2] *pat2bas*([*PC*]) = *app*(*list*, *tuple*(*pat2bas*(*PC*)))

[brackets-1] *pat2bas*((*PA*)) = *pat2bas*(*PA*)

[constructor-1] *pat2bas*(*op?* *LI* *AP1*) = *app*(*val*(*LI*), *pat2bas*(*AP1*))

[infix-1] *pat2bas*(*PA1* *I* *PA2*) =
app(*val*(*I*), *tuple*(*pat2bas*(*PA1*) *pat2bas*(*PA2*)))

[type-1] *pat2bas*(*PA* : *T*) = *pat2bas*(*PA*)

[as-1] *pat2bas*(*op?* *I* *CTY?* as *PA*) =
simult(*val-or-var*(*I*) *pat2bas*(*PA*))

[pat-seq-1] *pat2bas*(*AP1* *APS*) = *pat2bas*(*AP1*) *pat2bas*(*APS*)

[pat-seq-2] *pat2bas*(*APS*) = *patcast*(*AP1*) **when** *APS* = *AP1*

[cast-1] *patcast*(*PA*) = *pat2bas*(*PA*)

C.3 Declarations

In connection with the mapping of function declarations a lot of auxiliary functions are used to make a tuple of fresh identifiers (*make-id-tuple*), a chain of anonymous functions (*make-fn-chain*) and constructing the right match (*get-match* and *set-match*). The lexical constructors `valueid` and `natcon` are used to construct identifiers named '`vid_`i', where i is a positive integer. For more information about the mapping of function declarations consult Sect. 3.3

The following list shows the signatures of the functions used in the mapping to BAS.

dec2bas	:	*Declaration* → *Dec*
dec2bas	:	*ValueBinding* → *Dec*
dec2bas	:	*SingleFunValueBinding* → *Dec*
dec2bas	:	*SingleFunValueBindingBar* → *Dec*
dec2bas	:	*FunValueBinding* → *Dec*
deccast	:	*SingleFunValueBindingBar* → *Dec*
deccast	:	*SingleFunValueBinding* → *Dec*
dec2bas	:	*TypeBinding* → *Dec*
dec2bas	:	*SingleTypeBinding* → *Dec*
deccast	:	*SingleTypeBinding* → *Dec*
dec2bas	:	*DataBinding* → *Dec*
dec2bas	:	*ConsBinding* → *Dec*
deccast	:	*SingleDataBinding* → *Dec*
dec2bas	:	*SingleConsBinding* → *Dec*
deccast	:	*SingleConsBinding* → *Dec*
dec2bas	:	*ExcBinding* → *Dec*
dec2bas	:	*SingleExcBinding* → *Dec*
deccast	:	*SingleExcBinding* → *Dec*
numofargs	:	*SingleFunValueBindingBar* → *Integer*
length	:	*AtomicPattern+* → *Integer*
curry	:	*Exp, Integer* → *Exp*

The following list shows the variables used in the mapping to BAS and the syntactic sorts they range over.

D	:	*Declaration*	*TVS*	:	*TypeVarSequence*
TV	:	*TypeVariable*	*TV**	:	*TypeVariable**
TVC	:	{ *TypeVariable* "," }+	*EX*	:	*Expression*
P	:	*Pattern*	*VB*	:	*ValueBinding*
SVB	:	*SingleValueBinding*	*AD**	:	*Dec**
*AE**	:	*Exp**	*AE*	:	*Exp*
FVB	:	*FunValueBinding*	*SFVB*	:	*SingleFunValueBinding*
SFVBB	:	*SingleFunValueBindingBar*	*I*	:	*ValueId*
I+	:	*ValueId+*	*LI*	:	*LongValueId*
ATP+	:	*AtomicPattern+*	*ATP**	:	*AtomicPattern**
ATP	:	*AtomicPattern*	*AP*	:	*Par*
CTY?	:	(":" *Type*)?	*OTY?*	:	("of" *Type*)?
TY	:	*Type*	"op?"	:	"op"?
TB	:	*TypeBinding*	*STB*	:	*SingleTypeBinding*
STBS	:	{ *SingleTypeBinding* "and" }+	*TC*	:	*TypeConstructor*
LTC	:	*LongTypeConstructor*	*LS*	:	*LongStringId**
d?	:	*Digit?*	*DB*	:	*DataBinding*
WTB?	:	(withtype TypeBinding)?	*SDB*	:	*SingleDataBinding*
CB	:	*ConsBinding*	*SCB*	:	*SingleConsBinding*
EB	:	*ExcBinding*	*SEB*	:	*SingleExcBinding*
N	:	*NatCon*	*CH+*	:	*CHAR+*

equations

[val-1] $\mathit{dec2bas}(\texttt{val}\ \mathit{TVS}\ \mathit{VB}) = \mathit{dec2bas}(\mathit{VB})$

[val-2] $\mathit{dec2bas}(P = \mathit{EX}) = \textsf{bind-val}(\mathit{pat2bas}(P), \mathit{exp2bas}(\mathit{EX}))$

[val-3] $\mathit{dec2bas}(\mathit{SVB}\ \texttt{and}\ \mathit{VB}) = \textsf{simult-seq}(\mathit{dec2bas}(\mathit{SVB})\ \mathit{AD*})$
  when $\mathit{dec2bas}(\mathit{VB}) = \textsf{simult-seq}(\mathit{AD*})$

[val-4] $\mathit{dec2bas}(\texttt{rec}\ \mathit{VB}) = \textsf{rec}(\mathit{dec2bas}(\mathit{VB}))$

[default-val] $\mathit{dec2bas}(\mathit{SVB}\ \texttt{and}\ \mathit{VB}) = \textsf{simult-seq}(\mathit{dec2bas}(\mathit{SVB})\ \mathit{dec2bas}(\mathit{VB}))$

[fun-1]
$$\frac{\begin{array}{l}\mathit{SFVBB} = \mathit{FVB},\\ \textsf{bind-val}(\mathit{AP}, \mathit{AE1}) = \mathit{dec2bas}(\mathit{FVB}),\\ N = \mathit{numofargs}(\mathit{SFVBB}),\\ \mathit{AE2} = \textsf{app-seq}(\textsf{abs}(\textsf{val}(\texttt{valueid("f")}), \mathit{curry}(\textsf{tuple-seq}(), N)), \mathit{AE1})\end{array}}{\mathit{dec2bas}(\texttt{fun}\ \mathit{TVS}\ \mathit{FVB}) = \textsf{rec}(\textsf{bind-val}(\mathit{AP}, \mathit{AE2}))}$$

[default-fun-1] $\mathit{dec2bas}(\texttt{fun}\ \mathit{TVS}\ \mathit{FVB}) = \textsf{rec}(\mathit{dec2bas}(\mathit{FVB}))$

[fun-2]
$$\frac{\mathit{AE1} = \textsf{abs}(\textsf{tuple}(\mathit{pat2bas}(\mathit{ATP+})), \mathit{exp2bas}(\mathit{EX}))}{\mathit{dec2bas}(\mathit{op?}\ I\ \mathit{ATP+}\ \mathit{CTY?} = \mathit{EX}) = \textsf{bind-val}(\textsf{var}(I), \mathit{AE1})}$$

[fun-3]
$$\frac{\begin{array}{l} \mathit{dec2bas}(\mathit{SFVB}) = \textsf{bind-val}(\mathit{AP1},\ \mathit{AE1}), \\ \mathit{dec2bas}(\mathit{SFVBB}) = \textsf{bind-val}(\mathit{AP1},\ \mathit{AE2}) \end{array}}{\mathit{dec2bas}(\mathit{SFVB}\ \texttt{|}\ \mathit{SFVBB}) = \textsf{bind-val}(\mathit{AP1},\ \textsf{alt-seq}(\mathit{AE1}\ \ \mathit{AE2}))}$$

[fun-5]
$$\frac{\begin{array}{l} \textsf{bind-val}(\mathit{AP},\ \mathit{AE1}) = \mathit{dec2bas}(\mathit{SFVBB}), \\ N = \mathit{numofargs}(\mathit{SFVBB}), \\ \mathit{AE2} = \textsf{app-seq}(\textsf{abs}(\textsf{val}(\texttt{valueid("f")}),\ \mathit{curry}(\textsf{tuple-seq}(),\ N)),\ \mathit{AE1}) \\ \textsf{simult-seq}(\mathit{AD}*) = \mathit{dec2bas}(\mathit{FVB}) \end{array}}{\mathit{dec2bas}(\mathit{SFVBB}\ \texttt{and}\ \mathit{FVB}) = \textsf{simult-seq}(\textsf{bind-val}(\mathit{AP},\ \mathit{AE2})\ \ \mathit{AD}*)}$$

[default-fun-5]
$$\frac{\begin{array}{l} \textsf{bind-val}(\mathit{AP1},\ \mathit{AE1}) = \mathit{dec2bas}(\mathit{SFVBB}), \\ N1 = \mathit{numofargs}(\mathit{SFVBB}), \\ \mathit{AE2} = \textsf{app-seq}(\textsf{abs}(\textsf{val}(\texttt{valueid("f")}), \\ \qquad\qquad \mathit{curry}(\textsf{tuple-seq}(),\ N1)),\ \mathit{AE1}) \\ \textsf{bind-val}(\mathit{AP2},\ \mathit{AE3}) = \mathit{dec2bas}(\mathit{FVB}), \\ \mathit{SFVBB2} = \mathit{FVB}, \\ N2 = \mathit{numofargs}(\mathit{SFVBB2}), \\ \mathit{AE4} = \textsf{app-seq}(\textsf{abs}(\textsf{val}(\texttt{valueid("f")}), \\ \qquad\qquad \mathit{curry}(\textsf{tuple-seq}(),\ N2)),\ \mathit{AE3}) \end{array}}{\begin{array}{l} \mathit{dec2bas}(\mathit{SFVBB}\ \texttt{and}\ \mathit{FVB}) = \\ \qquad \textsf{simult-seq}(\textsf{bind-val}(\mathit{AP1},\ \mathit{AE2})\ \ \textsf{bind-val}(\mathit{AP2},\ \mathit{AE4})) \end{array}}$$

[fun-6] $\mathit{dec2bas}(\mathit{SFVBB}) = \mathit{deccast}(\mathit{SFVB})$ **when** $\mathit{SFVBB} = \mathit{SFVB}$

[fun-7] $\mathit{dec2bas}(\mathit{FVB}) = \mathit{deccast}(\mathit{SFVBB})$ **when** $\mathit{SFVBB} = \mathit{FVB}$

[numofargs-1] $\mathit{numofargs}(\mathit{op?}\ \ I\ \ \mathit{ATP+}\ \ \mathit{CTY?} = \mathit{EX}\ \texttt{|}\ \mathit{SFVBB}) = \mathit{length}(\mathit{ATP+})$

[numofargs-2] $\mathit{numofargs}(\mathit{op?}\ \ I\ \ \mathit{ATP+}\ \ \mathit{CTY?} = \mathit{EX}) = \mathit{length}(\mathit{ATP+})$

[length-1] $\mathit{length}(\mathit{ATP1}\ \ \mathit{ATP+}) = 1 + \mathit{length}(\mathit{ATP+})$

[length-2] $\mathit{length}(\mathit{ATP1}) = 1$

[default-curry-1]
$$\frac{\begin{array}{l} N = \texttt{natcon}(\mathit{CH+}), \\ \textsf{val}(I) = \textsf{val}(\texttt{valueid("v""i""d""\_"}\mathit{CH+})) \end{array}}{\begin{array}{l} \mathit{curry}(\textsf{tuple-seq}(\mathit{AE}*),\ N) = \\ \qquad \textsf{abs}(\textsf{var}(I),\ \mathit{curry}(\textsf{tuple-seq}(\mathit{AE}*\ \ \textsf{val}(I)),\ N\text{-}1)) \end{array}}$$

[curry-2] $\mathit{curry}(\mathit{AE},\ 0) = \textsf{app-seq}(\textsf{val}(\texttt{valueid("f")}),\ \mathit{AE})$

[cast-1] $\mathit{deccast}(\mathit{SFVB}) = \mathit{dec2bas}(\mathit{SFVB})$

[cast-2] $\mathit{deccast}(\mathit{SFVBB}) = \mathit{dec2bas}(\mathit{SFVBB})$

[type-1] $dec2bas$(type TB) = *ignore*

[datatype-1] $dec2bas$(datatype DB $WTB?$) = $dec2bas(DB)$

[datatype-3] $dec2bas(TVS\ TC$ = $CB)$ = $dec2bas(CB)$

[datatype-4] $dec2bas(SDB$ and $DB)$ = *simult-seq*($dec2bas(SDB)$ $AD*$)
when $dec2bas(DB)$ = *simult-seq*($AD*$)

[default-datatype-4] $dec2bas(SDB$ and $DB)$ =
simult-seq($dec2bas(SDB)$ $dec2bas(DB)$)

[datatype-5] $dec2bas(DB)$ = $deccast(SDB)$ **when** SDB = DB

[cast-4] $deccast(SDB)$ = $dec2bas(SDB)$

[cons-bind-1] $dec2bas(SCB)$ = *bind-val*($pat2bas(I)$, *new-cons*)
when SCB = $op?\ I\ OTY?$

[cons-bind-3] $dec2bas(SCB$ | $CB)$ = *simult-seq*($dec2bas(SCB)$ $AD*$)
when $dec2bas(CB)$ = *simult-seq*($AD*$)

[default-cons-bind-3] $dec2bas(SCB$ | $CB)$ =
simult-seq($dec2bas(SCB)$ $dec2bas(CB)$)

[cons-bind-4] $dec2bas(CB)$ = $deccast(SCB)$ **when** CB = SCB

[cast-5] $deccast(SCB)$ = $dec2bas(SCB)$

[datatype-6] $dec2bas$(datatype TC = datatype LTC) = *ignore*

[abstype-1] $dec2bas$(abstype DB $WTB?$ with $D1$ end) =
local($dec2bas(DB)$, $dec2bas(D1)$)

[exception-1] $dec2bas$(exception EB) = $dec2bas(EB)$

[exception-2] $dec2bas(SEB)$ = *bind-val*($pat2bas(I)$, *new-cons*)
when $op?\ I\ OTY?$ = SEB

[exception-4] $dec2bas(op?\ I$ = $op?\ LI)$ =
bind-val($pat2bas(I)$, $exp2bas(LI)$)

[exception-5] $dec2bas(SEB$ and $EB)$ = *simult-seq*($dec2bas(SEB)$ $AD*$)
when $dec2bas(EB)$ = *simult-seq*($AD*$)

[default-exception-5] $\mathit{dec2bas}$(*SEB* and *EB*) =
simult-seq($\mathit{dec2bas}$(*SEB*) $\mathit{dec2bas}$(*EB*))

[exception-6] $\mathit{dec2bas}$(*EB*) = $\mathit{deccast}$(*SEB*) **when** *SEB* = *EB*

[cast-6] $\mathit{deccast}$(*SEB*) = $\mathit{dec2bas}$(*SEB*)

[local-1] $\mathit{dec2bas}$(local *D1* in *D2* end) = *local*($\mathit{dec2bas}$(*D1*), $\mathit{dec2bas}$(*D2*))

[open-1] $\mathit{dec2bas}$(open *LS*) = *ignore*

[seq-1] $\mathit{dec2bas}$(*D1* ; *D2*) = *accum*($\mathit{dec2bas}$(*D1*) $\mathit{dec2bas}$(*D2*))

[seq-2] $\mathit{dec2bas}$(*D1* *D2*) = *accum*($\mathit{dec2bas}$(*D1*) $\mathit{dec2bas}$(*D2*))

[infix-1] $\mathit{dec2bas}$(infix *d?* *I+*) = *ignore*

[infix-2] $\mathit{dec2bas}$(infixr *d?* *I+*) = *ignore*

[infix-2] $\mathit{dec2bas}$(nonfix *I+*) = *ignore*

D Action semantic descriptions of BAS constructs

Module 1 *CoreML*

imports

Exp
Exp/Val
Exp/Val-Id-Const
Exp/App-Seq
Exp/Tuple-Seq
Exp/Cond
Exp/Abs
Exp/Alt-Seq
Exp/Seq-Stm-Exp
Exp/Throw
Exp/Catch
Exp/Local

Stm
Stm/Exp
Stm/While

Par
Par/Val
Par/Val-Or-Var
Par/Anon
Par/Var
Par/App
Par/Tuple
Par/Simult

Dec
Dec/Bind-Val
Dec/Simult-Seq
Dec/Rec
Dec/Local
Dec/Accum
Dec/Ignore

D.1 Expressions

Module 2 *Exp*

requires

E : *Exp*

Datum ::= *Val*

semantics evaluate: *Exp* → *Action*

Module 3 *Exp/Val*

syntax *Exp* ::= *Val*

semantics evaluate *V* = give the val *V*

Module 4 *Exp/Val-Id-Const*

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

requires *Val* ::= *Cons*

Cons ::= cons(val: *Val*)

Bindable ::= *Val*

semantics evaluate val(*I*) =
maybe give val(the cons bound-to the token *I*)
else give the val bound-to the token *I*

Module 5 *Exp/App-Seq*

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

requires *Val* ::= *Func*

func-no-apply : *Val*

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

Module 6 *Exp/Tuple-Seq*

syntax *Exp* ::= tuple-seq(*Exp*+)

requires *Val* ::= *Tuple*

semantics

evaluate tuple-seq(*E*) = evaluate *E* then give tuple(the val)

evaluate tuple-seq(*E* *E*+) =
evaluate *E* and-then
evaluate tuple-seq(*E*+) then
give tuple(the val#1,
components(the tuple#2))

Module 7 *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 8 *Exp/Abs*

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

requires *Val* ::= *Func*

semantics evaluate abs(*P*, *E*) =
give func(closure(furthermore match *P* scope evaluate *E*))

Module 9 *Exp/Alt-Seq*

syntax *Exp* ::= alt-seq(*Exp*+)

requires *Val* ::= *Func*

semantics

evaluate alt-seq(*E*) = evaluate *E* then give the func

evaluate alt-seq(*E* *E*+) =
evaluate *E* and-then
evaluate alt-seq(*E*+) then
give func(action(the func#1) else action(the func#2))

Module 10 *Exp/Seq-Stm-Exp*

syntax *Exp* ::= seq(*Stm*, *Exp*)

semantics evaluate seq(*S*, *E*) = execute *S* and-then evaluate *E*

Module 11 *Exp/Throw*

syntax *Exp* ::= throw(*Exp*)

semantics evaluate throw(*E*) = evaluate *E* then throw it

Module 12 *Exp/Catch*

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

requires *Val* ::= *Func*

semantics evaluate catch(*E1*, *E2*) =
evaluate *E1* catch
(evaluate *E2* and give the val
then apply (action(the func#1), the val#2)
else throw the val)

Module 13 *Exp/Local*

syntax *Exp* ::= local(*Dec*, *Exp*)

semantics evaluate local(*D*, *E*) =
furthermore declare *D* scope evaluate *E*

D.2 Statements

Module 14 *Stm*

requires S : *Stm*

semantics execute : *Stm* → *Action*

Module 15 *Stm/Exp*

syntax *Stm* ::= stm(*Exp*)

semantics execute stm(E) = evaluate E then skip

Module 16 *Stm/While*

syntax *Stm* ::= while(*Exp*, *Stm*)

requires *Val* ::= *Boolean*

semantics execute while(E, S) =
unfolding (evaluate E then
maybe check the boolean then
execute S then unfold
else skip)

D.3 Parameters

Module 17 *Par*

requires P : *Par*

semantics match : *Par* → *Action*

Module 18 *Par/Val*

syntax *Par* ::= *Val*

semantics match V = maybe check (the val = V) then give no-bindings

Module 19 *Par/Anon*

syntax *Par* ::= anon

semantics match anon = give no-bindings

Module 20 *Par/Val-Or-Var*

syntax *Par* ::= val-or-var(*Ide*)

requires *Val* ::= *Cons*

Cons ::= cons(val:*Val*)

Bindable ::= *Val*

semantics match val-or-var(*I*) =
(given val(the cons bound-to the token *I*)
then give no-bindings)
else
bind(the token *I*, the val)

Module 21 *Par/Var*

syntax *Par* ::= var(*Ide*)

requires *Bindable* ::= *Val*

semantics match var(*I*) = bind(the token *I*, the val)

Module 22 *Par/App*

syntax *Par* ::= app(*Exp*, *Par*)

requires

Val ::= *Func*

PrefixDataOp ::= invert

semantics match app(*E*, *P*) =
give the val and
evaluate *E* then
maybe give invert(the func#2, the val#1)
then match *P*

Module 23 *Par/Tuple*

syntax *Par* ::= tuple(*Par*+)

requires *Val* ::= *Tuple*

semantics

match tuple(*P*) =
    maybe give components(the tuple) then
    give the val then match *P*

match tuple(*P* *P*+) =
    maybe give components(the tuple) then
    ((give the val#1 then match *P*) and
     (give tuple(#-1) then match tuple(*P*+)))
    then give disj-union

Module 24 *Par/Simult*

syntax *Par* ::= simult(*Par*+)

semantics

match simult(*P*) = match *P*

match simult(*P* *P*+) = match *P* and match simult(*P*+) then give disj-union

D.4 Declarations

Module 25 *Dec*

requires *D* : *Dec*

semantics declare : *Dec* $\rightarrow$ *Action*

Module 26 *Dec/Bind-Val*

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

semantics declare bind-val(*P*, *E*) = evaluate *E* then match *P*

Module 27 *Dec/Simult-Seq*

syntax *Dec* ::= simult-seq(*Dec*+)

semantics

declare simult-seq(*D*) = declare *D*

declare simult-seq(*D* *D*+) =
declare *D* and-then
declare simult-seq(*D*+) then
give disj-union

Module 28 *Dec/Rec*

syntax *Dec* ::= rec(*Dec*)

semantics declare rec(*D*) = recursively declare *D*

Module 29 *Dec/Local*

syntax *Dec* ::= local(*Dec*, *Dec*)

semantics declare local(*D1*, *D2*) =
furthermore declare *D1* scope declare *D2*

Module 30 *Dec/Accum*

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

semantics

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

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

Module 31 *Dec/Ignore*

syntax *Dec* ::= ignore

semantics declare ignore = give no-bindings

D.5 Data

Module 32 *Data/Bindings*

requires *Sort* ::= bindings | token | bindable

Module 33 *Data/Boolean*

requires

true : *Boolean*

false : *Boolean*

Sort ::= boolean

Module 34 *Data/Cons*

requires

Sort ::= cons

Val ::= new-cons

Module 35 *Data/Func*

requires *Func* ::= func(action: *Action* & using val & giving val)

Module 36 *Data/Tuple*

requires *Tuple* ::= tuple(components : *Data*)

List of Tables

List of Figures

Recent BRICS Report Series Publications

RS-04-37 **Jørgen Iversen and Peter D. Mosses.** ***Constructive Action Semantics for Core ML*****. December 2004. 68 pp. To appear in a special *Language Definitions and Tool Generation* issue of the journal *IEE Proceedings Software*.**

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.**