Basic Research in Computer Science

Formalisms and tools supporting Constructive Action Semantics

Jørgen Iversen

BRICS Dissertation Series **DS-05-2**

ISSN 1396-7002 **May 2005**

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

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/05/2/`

Formalisms and tools supporting Constructive Action Semantics

Jørgen Iversen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Formalisms and tools supporting Constructive Action Semantics

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Jørgen Iversen
February 22, 2005

Abstract

This dissertation deals with the problem of writing formal semantic descriptions of programming languages, that can easily be extended and partly reused in other descriptions. Constructive Action Semantics is a new version of the original Action Semantics framework that solves the problem by requiring that a description consists of independent modules describing single language constructs. We present a formalism and various tools for writing and using constructive action semantic descriptions of programming languages.

In part I we present formalisms for describing programming languages. The Action Semantics framework was developed by Mosses and Watt and is a framework for describing the semantics of real programming languages. The ASF+SDF formalism can among other things be used to describe the concrete syntax of a programming language and a mapping to abstract syntax, as we illustrate in a description of a subset of the Standard ML language. We also introduce a novel formalism, ASDF, for writing action semantic descriptions of single language constructs.

Part II of the dissertation is about tools that supports writing constructive action semantic descriptions and generating compilers from them. The Action Environment is an interactive development environment for writing action semantic descriptions using ASF+SDF and ASDF. A type checker available in the environment can be used to check the semantic functions in the ASDF modules. Actions can be evaluated using an action interpreter we have developed, and this is useful when prototyping a language description. Finally we present an action compiler that can be used in compiler generation. The action compiler consists of a type inference algorithm and a code generator.

Acknowledgements

A PhD dissertation is not only a single persons work. Many people have contributed with help and support in various ways.

First of all I would like to thank my supervisor Peter D. Mosses for suggesting this project. During my PhD study he has raised many problems and also helped solve a great deal of them. Also his perfectionism and endurance have been beneficial for me.

During my half year stay at CWI in Amsterdam, I came to know a lot of fine people in the SEN1 group. I am grateful for their hospitality and good friendship. My host, Mark van den Brand, and also Hayco de Jong, Paul Klint, and Jurgen Vinju, made it a pleasant and productive stay in Amsterdam.

Thanks to Janus Dam Nielsen, Jan Midtgaard, and Fabricio Chalub for proofreading some of my papers.

I also wish to thank the other PhD students (non mentioned, non forgotten), the secretaries, and the professors at BRICS for creating a good and friendly working environment.

Finally I would like to thank my wife for bearing with me in the periods where I have spent more time with a computer than with her. Thanks for supporting me when work was wearing me down.

Jørgen Iversen,
Århus, February 22, 2005.

Contents

Chapter 1

Introduction

Computer language design is just like a stroll in the park.
Jurassic Park, that is.
— Larry Wall

Formal descriptions are essential for giving precise definitions of programming languages. A language description must describe both the syntax and the semantics of a language to be complete. For describing the syntax, BNF grammars are popular both in academia and industry, for instance, Sun and Microsoft document the syntax of Java and C# using BNF grammars. Unfortunately the use of semantics seems to be limited in practical software development, and is widely regarded as only of academic interest [64]. We believe that formal descriptions of syntax have become popular because one formalism, BNF grammars, has become a dominating standard, whereas many semantic formalisms exist and are in use. Furthermore BNF grammars are easy to understand, whereas many semantic frameworks require deep mathematical insight, and use incomprehensible notation. Finally there is a broad selection of tools [14, 42, 78, 81] that generate parsers based on different classes of BNF grammars.

Tools that employ semantic descriptions for generating language analysers and compiler back-ends also exist [23, 25, 36, 49, 54, 79, 94], (see [48, 86] for an overview), but most of them seem to be academic projects that have never found any use in the real world.

Action Semantics (AS) [45,56–59,65,67,68,70,89] is a framework for describing the semantics of real programming languages. It uses a notation (Action Notation) that resembles English which makes it easy to read, and previous work has shown that it can be used in semantics-based compiler generation.

This dissertation presents a set of tools that support the process of writing an action semantic description of a programming language, checking the validity of the description, testing the description, and automatically generating a compiler from the description. Our work is a step towards a widespread use of semantics and semantics-based compiler generation.

Writing the description of a language is supported by a novel formalism, ASDF, for writing *constructive action semantic descriptions*. Constructive Action Semantics is a new version of the original AS framework that supports the independent definition and use of named modules describing individual

language features[1]. A module describing a single language feature should be reusable by import in many descriptions. This is possible by keeping the modules describing different features independent and by using a language-independent abstract syntax in the modules. We have developed a set of modules in connection with a description of the Core of Standard ML. The set of modules is referred to as *Basic Abstract Syntax* (BAS), and the idea is that they should be general enough to be reused in other language descriptions. New constructs can be defined in ASDF and used together with the existing BAS constructs when describing a language.

In connection with the ASDF formalism we present the Action Environment, an environment for working with language descriptions written in ASDF and ASF+SDF [14].

Checking the validity of a language description can be done using a semantic function type checker, which lets us check that the output of a semantic function is an action that has certain features, like being free of side effects, infallible, etc. For testing a language description, an action interpreter is provided which can be used to evaluate the result of mapping a program to an action using the semantic functions. Both the type checker and the action interpreter are connected to the Action Environment.

1.1 Action Semantics-based compiler generation

Action Semantics-based compiler generation has been an area of research for almost 15 years. The Actress system was one of the first systems [23, 73]. With the Cantor system [76] Palsberg showed how to generate provably correct compilers. Ørbæk developed the OASIS system [94] which produced highly efficient compilers. Bondorf and Palsberg [6] showed how partial evaluation can be used to generate a compiler based on an action interpreter. The Genesis system [46], developed by Lee, improved the techniques used in the Actress system.

Our compiler generator uses the Action Environment to generate the front end of a compiler as illustrated in Fig. 1.1. Input for the compiler generator is shown as circles, existing tools as rectangles, generated tools as rectangles with round corners, and input and intermediate representations in the compiler as ellipses. The input for the compiler generator consists of SDF, used to describe the syntax of the language, ASF, used to describe a mapping from the syntax of the language to BAS, and ASDF, used to describe the syntax and AS of the BAS constructs. From the input the Action Environment can generate a front end consisting of a parser, a translator from concrete syntax to BAS, and an action generator that maps BAS to actions. The back-end of the compiler is an action compiler based on our action type inference algorithm and our code generator. The action compiler also uses the ASDF input when doing type inference and code generation. The structure of our compiler generator is similar to the one used in previous work [23, 46, 94].

[1] See Section 1 in [41] for an elaborate explanation of constructive semantic frameworks.

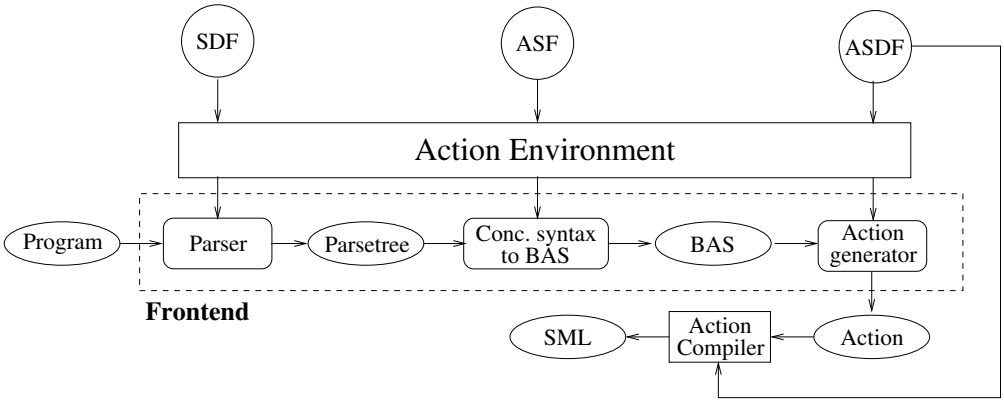

Figure 1.1: AS-based compiler generation

1.2 Organisation

The contents of this dissertation is both material published in journals and at workshops and material that have not been published before. The contents of the following papers is used in the dissertation:

[12] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. Research Series BRICS RS-04-36, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2004. Extended version of [13], submitted to a special issue of *Science of Computer Programming* for LDTA'04 papers.

[38] J. Iversen. Type inference for the new action notation. In Mosses [65], pages 78–98.

[39] J. Iversen. Type checking semantic functions in ASDF. Research Series BRICS RS-04-35, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2004. `http://www.brics.dk/RS/04/35/`.

[41] J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *IEE Proceedings-Software special issue on Language Definitions and Tool Generation*, 2005, to appear.

[40] J. Iversen. An action compiler targeting Standard ML. In J. Boyland and G. Hedin, editors, *Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications, LDTA'05*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005, to appear. Also available at `http://www.daimi.au.dk/~jive/papers/ldta2005.pdf`.

Part I of the dissertation describes the formalisms used for expressing the input given to the tools presented in Part II. In Chapter 2 we give an informal but thorough description of AS and the syntax and semantics of Action Notation. Together with ASF+SDF, AS is one of the prerequisites for understanding the rest of the dissertation. ASF+SDF is explained in Chapter 3 where we also

describe the advantages and disadvantages of ASF+SDF and evaluate its usefulness as implementation language for the algorithms presented in the rest of the dissertation. The development of the ASDF formalism was driven by the inconvenience in using ASF+SDF for writing Action Semantic Descriptions of single language constructs. ASDF is motivated and described in Chapter 4.

To conclude Part I of the dissertation, we give a large example that illustrates how AS, ASF+SDF, and ASDF can be used to describe a real programming language. The Core ML example in Chapter 5 is taken from [41].

Part II presents the tools we have developed. The Action Environment is presented in Chapter 6 (based on [12]). Chapter 7 presents the action types that can be used to describe the runtime behaviour of an action, along with an algorithm for checking that an action conforms to an action type. Chapter 7 is based on [39]. The action interpreter is described in Chapter 8.

An important component of an AS-based compiler generator is the action compiler. Our action compiler consists of the type inference algorithm described in Chapter 9 (a revised version of [38]) and the code generator described in Chapter 10 (based on [40]).

Chapter 11 concludes and outlines directions for future work.

Part I

Formalisms

Chapter 2

Action Semantics

The superior man is modest in his speech, but exceeds in his actions.
— Confucius

Action Semantics (AS) is a framework for describing the semantics[1] of real programming languages. The inherent modularity makes it easy to maintain and extend descriptions. AS is a hybrid of Denotational Semantics and Operational Semantics. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations which model their behaviour. The difference is that here denotations are *actions* and expressed in *Action Notation* (AN) [29, 45, 56, 66]; a notation resembling English but still strictly formal. AN has a *kernel* which is defined operationally; the rest of AN is abbreviations for other actions and can be reduced to kernel notation. Actions are constructed from *yielders*, action constants, and action combinators, where yielders consist of data and data operations. Yielders are not part of the kernel.

AN has changed several times since Mosses [53] started experimenting with combinators as auxiliary notation in Denotational Semantics. The version used in this paper is the one presented in *Definitive Semantics* [66]. We shall sometimes refer to this version as AN-2 to explicitly state that it is the new version. AN-2 has evolved over the last five years. The main differences between the older versions and the version proposed in 2000 [45] is that actions no longer produce both data and bindings, and the kernel has been reduced in size.

The performance of an action might be seen as an evaluation of a function from data and bindings to data, with side effects like changing storage and sending messages. Action combinators correspond to 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 (Section 2.3), declarations (Section 2.4), manipulation of storage (Section 2.5), and abstractions

[1] AS is mostly used to describe a language's dynamic semantics, but has also been used to describe a language's static semantics [35]

(Section 2.6). It is also possible to describe the special features of concurrent languages using AN, but we are not going to deal with these actions in this dissertation, and they are therefore not presented in this chapter. Yielders can be used to inspect memory locations, compute data and bindings, and to lookup bound tokens. As with actions, the performance of a yielder can be seen as an evaluation of a function from data and bindings to data, but without side effects.

In this dissertation we shall use the variables A to range over actions, Y to range over yielders, D to range over data, and DO to range over data operations. The phrases "the given data" and "the given bindings" (or "the current bindings") are used to refer to the data input and the bindings input for an action, respectively.

2.1 Action semantic descriptions

An *action semantic description* (ASD) of a programming language must describe the abstract syntax of the language, the semantic functions mapping language constructs to actions, and data, data operators, types, etc., occurring in the semantic functions.

The semantic equation that defines the semantic function's behaviour on a specific language construct might contain applications of semantic functions to subparts of the language construct; semantic functions are defined recursively. Here is an example of a semantic equation that defines the semantic function *evaluate* on the expression '*E1* + *E2*':

evaluate(*E1* + *E2*) = *evaluate E1* and *evaluate E2* then give +

When applying a semantic function to a program (or a language construct), the result should be an action with the same dynamic semantics as the program (or the language construct). The resulting action should describe the operational behaviour of the program.

Among the types being described by the user there should be a description of the two types *Bindable* and *Storable*. The data that can be bound to tokens in bindings should have the type *Bindable*, and the data that can be stored in memory cells should have the type *Storable*.

2.2 Data and data operations

The data used and produced by actions is the union of the data defined by the user and the data described by the grammar in Fig. 2.1. *Data* is a sequence of *Datum*. We shall use tuple notation when describing data in actions, and the empty sequence of datum is represented by the symbol (). The non-terminals used to describe *Datum* expand to integer numbers, the boolean values true and false, tokens, memory locations (or memory cells), finite mappings from tokens to datum (also called *bindings*), and finally actions (more about actions as data in Section 2.6). We shall assume that *Token* contains the identifiers

used in the programming language examples given in the dissertation together with positive integers.

Data	::=	*Datum**
Datum	::=	*Integer* | *Boolean* | *Token* | *Cell* | *Bindings* | *Action*

Figure 2.1: Data defined in AN

Floating point numbers, strings, and other data common in programming languages are not part of AN, but, as mentioned earlier, the data used in AN can be defined by the user.

The built-in data operations in AN are presented in Fig. 2.2. The first line includes the equality operator which compares data and results in a boolean. The boolean negation operator 'not' has the expected semantics. The type projector 'the τ' is a partial operator that on data of type τ behaves like the identity function. The selector operation # n selects the n'th element from a sequence of datum. If n is -1 the result is all elements except the first.

DataOp	::=	= | not | the *Type* | # *Natural* | # -1 |	(basic)
		+ | - | * | / | < | > |	(arithmetic)
		binding | bound | overriding | disj-union |	(bindings)
		ActionComb	(actions as data)

Figure 2.2: Data operators defined in AN

The next line in Fig. 2.2 contains the normal integer arithmetic operators. Line three contains operators for creating a singleton binding map (binding : *Token* × *Bindable* → *Bindings*), looking up a binding (bound : *Bindings* × *Token* →? *Bindable*)[2], computing bindings from two others where the second set of bindings has precedence over the first if the same token is bound in both sets (overriding : *Bindings* × *Bindings* → *Bindings*), and, finally, an operator for computing the disjoint union of two maps of bindings (disj-union : *Bindings* × *Bindings* →? *Bindings*).

The last line of operators contains the non-terminal *ActionComb* which expands to all the action combinators presented in the following sections. Given one or two actions as data (depending on whether it is a prefix or an infix action combinator), the action combinator computes a new action.

Later in this chapter we shall refer to the infix data operators (=, the arithmetic operators, and the infix action combinators) by the non-terminal *InfixDataOp* and the prefix data operators (the rest of the operators) by *PrefixDataOp*.

[2]The special arrow '→?' indicates that an operator is partial

2.3 Flow of data and control

Basic language constructs like expressions and sequential, conditional, and iterative statements can be described using the actions presented in this section. Exclusively using the grammars in Fig. 2.3 and 2.5 to construct actions results in actions that normally compute a value and do not have side effects (describing statements using only these actions is therefore not very interesting).

Action ::= copy | result *Data* | choose-nat | give *DataOp* |
Action then *Action* | *Action* and *Action* |
Action and-then *Action* | indivisibly *Action* |
throw | check *DataOp* |
Action catch *Action* | *Action* and-catch *Action* |
fail | *Action* else *Action* |
unfold | unfolding *Action*

Figure 2.3: Flow of data and control kernel AN

2.3.1 Normal control flow

Let us start by looking at the actions that describe normal control flow:

copy is essentially the identity function; it gives the data it is given as input and ignores the bindings. It is often used in connection with and (explained later) to add extra data to the sequence of data given to the whole action.

result D ignores the given data and bindings and gives the data D. It is used to describe constants, like numbers, identifiers, etc., in languages.

choose-nat produces a random non-negative integer.

give DO applies the data operator DO to the given data and ignores the bindings. If DO gives a result, the action terminates normally with the result, and otherwise it terminates abruptly with no data. Application of built-in data operators, like addition, in languages is usually described using give DO.

The four preceding actions were all atomic actions; now let us take a look at some ways to combine actions:

A_1 **then** A_2 is the normal function composition of A_1 and A_2. If applying A_1 to the given data and bindings terminates normally, the result is given to A_2 together with the given bindings, and the result of the whole action is the result of evaluating A_2. If A_1 does not terminate normally, A_2 is not evaluated, and the result of the whole action is the result of evaluating A_1. then is the most used action combinator.

A_1 **and** A_2 is an interleaving evaluation of the two subactions, both applied to the given data and bindings. If both subactions terminate normally, the whole action terminates normally with the concatenation of the data given by the two subactions. Otherwise the subaction that first terminates abruptly or failing determines the result of the whole action. This action combinator can be used

to describe the evaluation of the two subexpressions in applications of infix data operators when the order of evaluating the subexpressions is indifferent.

There are two actions directly related to 'A_1 and A_2':

A_1 **and-then** A_2 is the sequential version of 'A_1 and A_2'. A_1 is always evaluated before A_2, but except from this their behaviour is identical.

indivisibly A prohibits interleaving of A. The action A is evaluated with the data and bindings given to the whole action, but if 'indivisibly A' occurs as a subaction of an action constructed by the and combinator, the evaluation of A cannot be interleaved with the evaluation of other subactions. This allows the description of atomic events which is important when describing concurrent languages.

We now have enough notation established to give a meaningful example:

```
(result x and copy) then give binding
```

Assume that the action is given a bindable datum d. The two subactions 'result x' (which gives the token x) and copy (resulting in the datum d given to the whole action) is evaluated interleaved. The action combinator and concatenates the two results, and the combinator then ensures that 'give binding' is given (x, d), and the final result is the singleton binding map $\{\mathsf{x} \mapsto d\}$.

2.3.2 Abrupt control flow

Many languages have constructs to describe abrupt flow of control, e.g., `raise` and `handle` in Standard ML [51]. The following actions can be used to describe abrupt control flow:

throw terminates abruptly with the data given to it, ignoring bindings. It is used to describe raising exceptions and often used in connection with either catch or and-catch. The action throw is the abruptly terminating version of copy.

check DO applies the data operator DO to the given data, and if the result is true, the action terminates normally with the given data as result. Otherwise it terminates abruptly with no data. When describing conditional or guarded expressions or statements, this action is used.

A_1 **catch** A_2 first applies A_1 to the given data and bindings. If it terminates abruptly, the "exception" is "caught", and the resulting data together with the given bindings is given to A_2, which determines the result of the whole action. Otherwise the result of evaluating A_1 is the result of evaluating the whole action. Notice the similarity in behaviour with the combinator then; the only difference is that the left subaction should terminate abruptly instead of normally to trigger the evaluation of the right subaction.

A_1 **and-catch** A_2 can be related to A_1 and-then A_2 as catch to then. The explanation of and-catch is identical to the explanation of and-then (or and without interleaving) with "normally" replaced by "abruptly".

The action

(check the boolean) catch (result false then raise)

checks whether the input is the boolean value true. If it is, the action terminates normally with the result true. If it is not, it raises an exception with no data, the exception is caught, and a new exception is raised with the boolean value false.

2.3.3 Alternative control flow

Besides terminating normally and abruptly actions may also fail. This is used to describe a choice between alternatives; if the first fails, try the next.

fail simply ignores input and fails.

A_1 **else** A_2 evaluates A_1 with the given data and bindings, and if it fails, A_2 is evaluated with the same data and bindings. The result of evaluating the whole action is either the result of evaluating A_1 if it does not fail, or the result of evaluating A_2. Examples of language constructs that can be described using failing actions are alternatives as seen in C's `case` statement [43] or Standard ML's matches.

One might argue that we do not need both actions that can terminate abruptly and actions that can fail. Using special tokens to indicate special kinds of abrupt termination we might be able to describe the same behaviour. Practical experience has shown that it is convenient with two kinds of abrupt control flow. As an example take a description of conditional expressions:

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

where *evaluate* is the semantic function mapping expressions to actions, and *E1*, *E2*, and *E3* are variables ranging over expressions. If the expressions can throw exceptions (like in Standard ML), the else in the last line must not catch the exception because the exception should escape the whole expression. The else combinator should instead catch the failure caused by 'maybe check the boolean'. Here it is convenient with two ways of describing exceptional control flow. Failing termination can be described in terms of abrupt termination as illustrated in Fig. 2.4 (where FAIL and EXCEP are special tokens). But since the complexity of the expansion of the else combinator would make it difficult for a tool to infer the intention of the kernel action, else is still part of the AN kernel.

2.3.4 Iterative control flow

To describe iterative control flow, like while statements in Java, AN has two actions:

unfolding A evaluates the action A with the given data and bindings.

fail	⇒	raise FAIL
A_1 else A_2	⇒	(A_1 catch (result EXCEP and check (not(it = FAIL)))) and-catch A_2) then (((check not(#1 = EXCEP))) and-catch raise #-1)

Figure 2.4: fail and else as abbreviations

unfold evaluates the action A in the smallest enclosing 'unfolding A' (unfold is a subaction in A) with the data and bindings given to unfold. Together these two actions can describe iteration because evaluation of the action A is iterated as long as the contained unfold is evaluated.

One might regard 'unfolding A' as a way of declaring a recursive function with body A, and unfold as an application of the recursive function to the given data and bindings.

2.3.5 Abbreviations

To make ASDs easier to read and write, AN contains a number of abbreviations for kernel actions. Those related to flow of data and control are listed in Fig. 2.5.

Action	::=	give *Yielder* | check *Yielder* | *Action Yielder* | skip | maybe *Action*
Yielder	::=	*Data* | *DataOp* | (*Yielder*, ..., *Yielder*) | *PrefixDataOp Yielder* | *Yielder InfixDataOp Yielder*

Figure 2.5: Flow of data and control AN

Yielders are used to compute data without causing any side-effects. All data and data operators are included in yielders together with the data that can be computed by applying data operations to yielders and forming tuples of yielders. Tuples of yielders evaluate to sequences of data composed of the data resulting from each sub-yielder.

Three actions involving yielders are introduced in Fig. 2.5. The actions 'give Y' and 'check Y' have a semantics similar to 'give DO' and 'check DO', respectively (actually the latter are special cases of the former): instead of applying a data operator to the given data a yielder is applied to it (but yielders can also use the current bindings as shown in Section 2.4). The action 'A Y' evaluates the yielder Y, and the result is used as input in the evaluation of the action A.

skip ignores input and produces no data. It is used to end actions that are not supposed to give any data, like actions describing statements.

maybe A is used to turn an action that terminates abruptly into an action that fails. It can be used in connection with check DO and else to describe conditional statements.

The expansion to kernel AN is defined recursively in Fig. 2.6, where expansions of yielders are shown in the context of the 'give Y' action. We end this section with an example illustrating the use of yielders:

```
give (true, 2 + the integer)
then (maybe check (the integer#2 = 7) then skip
else choose-nat)
```

The first line results in the boolean value true and the sum of the given integer and 2. This data is given to an action that checks if the integer equals 7 and then terminates normally with no data. If the integer is not 7, maybe ensures that the action fails, and the alternative action in the next line is performed. The alternative action results in normal termination with a random integer as result.

give D	$\Rightarrow$	result D
give $(Y_1, \ldots, Y_n)$	$\Rightarrow$	give Y_1 and ... and give Y_n
give $(DO\ Y)$	$\Rightarrow$	give Y then give DO
give $(Y_1\ DO\ Y_2)$	$\Rightarrow$	give (Y_1, Y_2) then give DO
check Y	$\Rightarrow$	copy and (give Y then check it then skip)
$A\ Y$	$\Rightarrow$	give Y then A
skip	$\Rightarrow$	result ()
maybe A	$\Rightarrow$	A catch fail

Figure 2.6: Expansion to kernel AN

2.4 Scopes of bindings

Naming various parts of a program makes programming easier because it allows computed values, statement sequences, type definitions, etc. to be reused. In most languages the binding of an identifier to a value, a memory cell, a class, a function, etc., is called a *declaration*. This section concerns the actions used to describe declarations and the scopes of these.

Fig. 2.7 displays the three kernel actions used to describe declarations.

Action	::=	copy-bindings | *Action* scope *Action* | recursively *Action*

Figure 2.7: Scopes of bindings kernel AN

copy-bindings ignores the given data and gives the current bindings as data. This is used together with the data operator bound introduced in Fig. 2.2 to look up the datum bound to a token.

A_1 **scope** A_2 first evaluates A_1, and if it terminates normally and gives bindings, these are the current bindings when A_2 is evaluated with the data given to the whole action. If A_1 does not produce bindings, the whole action terminates

abruptly with no data. If A_1 terminates abruptly or failing, this is the result of the whole action. The combinator is used when describing a local scope of bindings; the scope of the bindings computed in A_1 is A_2.

recursively A lets the bindings produced by A override the bindings given to the whole action before evaluating A and thereby allows self-referential declarations in A.

The abbreviations are listed in Fig. 2.8, and the expansion to kernel notation can be found in Fig. 2.9.

Action	::=	furthermore *Action* | *Action* before *Action* | bind
Yielder	::=	bound-to *Yielder* | current-bindings

Figure 2.8: Scopes of bindings AN

furthermore A evaluates the action A with the given data and bindings, and if it terminates normally and produces bindings, the result of the whole action is the result of letting these bindings override the current bindings.

A_1 **before** A_2 first evaluates A_1 with the given data and bindings. The bindings b_1 produced by A_1 are combined with the given bindings b_0 (the former overrides the latter) to form the bindings b_1/b_0 given to A_2. The data produced by the whole action is the bindings b_1, produced by A_1, overridden by the bindings b_2 (b_2/b_1), produced by A_2. The rather complicated semantics of this action is reflected in the relatively large action it abbreviates. Notice that A_2 is given (b_0, b_1) as input data; a more natural input would be the data given to the whole action, but this would make the kernel action it abbreviates overly complicated, and often A_2 simply ignores its input. The action combinator before is used to describe a sequence of declarations where a declaration is visible to all the succeeding declarations.

bind expects a token and a bindable datum as input and produces the binding of the token to the datum.

Two yielders are available when describing bindings: 'bound-to Y' looks up the token which is the result of evaluating Y in the current bindings, and current-bindings evaluates to the current bindings.

furthermore A	⇒	copy-bindings and A then give overriding
A_1 before A_2	⇒	(copy-bindings and A_1) then (give #2 and (give overriding scope A_2)) then give overriding
bind	⇒	give binding
give bound-to Y	⇒	copy-bindings and give Y then give bound
give current-bindings	⇒	copy-bindings

Figure 2.9: Expansion to kernel AN

An example that uses bindings is

```
furthermore bind (x, 5)
scope give (the integer bound-to the token x)
```

where the first line produces a singleton binding map consisting of the token x bound to 5. This binding overrides the current bindings, and the result becomes the current bindings in the evaluation of the next line, where the binding of the token x is looked up.

To be able to give an Modular Structural Operational Semantic (MSOS) definition of 'recursively A' Mosses has suggested (this is not yet documented in a paper) to change the semantics so that 'recursively A' should be given a sequence of tokens. The tokens should be the tokens bound in the bindings given by A. The action 'recursively A' then binds the given tokens to a special kind of cells called *forwards* and gives these bindings together with the current bindings to A. Forwards are included in the set of bindable values and can be set to refer to bindable values. The contents of a forward is initially undefined, it can only be set once and it can never be destroyed. This change also influences the behaviour of bind and 'bound-to Y'. If bind receives a token which is bound to an undefined forward in the current bindings, it should set the forward to the given bindable value and give the bindings that map the token to the forward. In other cases, where the token is currently bound to a defined forward or a normal bindable value, bind behaves normally. If Y yields a token that is currently bound to a forward, 'bound-to Y' should yield the bindable value (if any) to which the forward has been set, else 'bound-to Y' behaves as described earlier.

In this dissertation we shall stick to the interpretations of 'recursively A', bind, and 'bound-to Y' presented earlier in this section, but the examples and tools can easily be changed to accommodate the new interpretation.

2.5 Effects on storage

So far we have only looked at actions that are purely functional; the actions already presented do not have any side-effects. In this section we present actions that can be used to manipulate storage.

In Fig. 2.10 the kernel actions are listed.

Action	::=	create | inspect | update

Figure 2.10: Effects on storage kernel AN

create allocates a fresh memory cell, initialises it with the storable datum given to create, and returns the cell. Together with the action bind from the previous section, create can be used to describe the declaration of variables in procedural programming languages.

inspect returns the datum stored in the given memory cell. Describing the use of variables in expressions typically involves inspect.

update expects a memory cell and a storable datum. The contents of the memory cell is replaced by the datum. This is used to describe assignment statements in procedural languages.

There is only one abbreviation related to manipulation of storage, and that is the yielder '**stored-at** Y' shown in Fig. 2.11. The yielder returns the datum stored at the memory cell computed by Y. The expansion to kernel notation is shown in Fig. 2.12

Yielder	::=	stored-at *Yielder*

Figure 2.11: Effects on storage AN

give stored-at Y	⇒	give Y then inspect

Figure 2.12: Expansion to kernel AN

The action

```
result 5 then create
then update (the cell, (the integer stored-at the cell) + 1)
```

allocates a new integer memory cell and initialises it with the integer 5. In the second line the contents of the memory cell is incremented.

2.6 Actions as data

The inclusion of actions in data turns actions into higher-order functions; actions become first-class values. This is useful for describing language constructs like functions and procedures.

Fig. 2.13 shows the two actions available to handle actions as data.

Action	::=	apply | close

Figure 2.13: Actions as data kernel AN

apply expects an action together with some data and evaluates the action with the data as input. The result of evaluating the action is the result of evaluating apply. This can be used to describe the application of a function or a procedure in a program.

close forms a closure from the given action, i.e., it ensures that the current bindings at the time where close is performed are available to the given action when it is later evaluated. This action is used both in descriptions of languages with static scope and languages with dynamic scope. When the scope is statically decidable, this action is applied when a function is declared to ensure

that the free variables in the function is bound to the same values at evaluation time as at definition time. In languages with dynamic scope close is used just before a function is applied so that the current bindings at invocation time can be used in the function.

Yielder	::=	closure *Yielder*

Figure 2.14: Actions as data AN

In connection with evaluating actions and forming closures of them, we also want to be able to produce actions as data. This does not require any new AN because we already have actions for giving data (result D), and actions are included in data. Since most languages have static scope, we often want to form a closure from an action, and therefore a yielder for doing just that exists. The yielder is shown in Fig. 2.14, and, not surprisingly, the expansion to kernel notation uses the close action (See Fig. 2.15).

give closure Y	⇒	give Y then close

Figure 2.15: Expansion to kernel AN

We can also form new actions at runtime by applying action combinators as data operators to actions. In a previous version of AN-2 [45] this was used to describe 'closure Y' (close was not part of the kernel) in the following way:

give closure Y	⇒	(copy-bindings then give result_) and give Y then give _scope_

In the action

```
give closure(give (the integer + the integer bound-to the token x))
then apply(the action, 2)
```

the closure of an action, that adds the integer bound to the token x to the given integer, is computed, and in the next line it is applied to the integer 2. The final result depends on what x is bound to in the in the bindings given to the whole action.

2.7 Parsing

To avoid ambiguous readings of an action without having to insert many brackets, we shall use some precedence rules when parsing actions. Fig. 2.16 shows the part of the AN syntax that is problematic. The non-terminals *InfixAction* and *PrefixAction* expand to the action combinators presented in the previous sections.

Brackets around actions and yielders are legal ways of disambiguating an action, but if they are not there, the following rules can be used:

Action	::=	*Action InfixAction Action* | *PrefixAction Action* | *Action Yielder* | give *Yielder* | check *Yielder*
Yielder	::=	*Yielder InfixDataOp Yielder* | *PrefixDataOp Yielder* | bound-to *Yielder* | stored-at *Yielder*| closure *Yielder*

Figure 2.16: AN syntax

1. Prefix action constructors (*PrefixAction*), like indivisibly, maybe, etc., have higher precedence than infix action constructors (*InfixAction*), like then, scope, etc.

2. Prefix data operators, like not, the *Type*, etc., have higher precedence than infix data operators, like +, >, etc.

3. All infix action constructors and infix data operators are left associative, and have the same precedence.

4. The actions *Action Yielder*, give *Yielder*, and check *Yielder* have higher precedence than *PrefixAction Action*.

We shall insist that the yielders occurring in '*Action Yielder*', 'give *Yielder*', and 'check *Yielder*' are atomic yielders. By atomic we mean data (with brackets around if it is an action), a data operator, or a bracketed yielder.

2.8 Advantages and disadvantages of AS

AS has several advantages. Most prominent is the inherent extensibility. From the AS website[3]:

> Action Semantics is a framework for the formal description of programming languages. Its main advantage over other frameworks is pragmatic: action-semantic descriptions (ASDs) scale up smoothly to realistic programming languages. This is due to the inherent extensibility and modifiability of ASDs, ensuring that extensions and changes to the described language require only proportionate changes in its description. (In Denotational or Operational Semantics, adding an unforeseen construct to a language may require a reformulation of the entire description.)

The claims about Denotational and Operational Semantics not being extensible are only true for the classic versions. In [63] Mosses gives an overview of modular semantic frameworks, among these is MSOS [60,61] and Monadic Denotational Semantics [47,52]. Both of them are modular and extensible versions of the classic frameworks.

[3] `http://www.brics.dk/Projects/AS/AboutActionSemantics.html`

ASDs are modular and extensible as long as we do not want to add any unusual constructs to the description. By unusual constructs we mean language constructs that cannot easily be described in AS. We can describe the semantics of all Turing complete languages since AN is Turing complete, but it is not obvious how we can easily describe, for instance, a construct like `call/cc` found in Scheme [1]. The standard example that demonstrates the extensibility of AS is an expression language where we want to add exceptions. This can easily be achieved without modifying the constructs already described since AN supports exceptional behaviour, and in general AN supports most constructs found in programming languages. The problem arises if we want to describe constructs not supported by AN.

Comparing AS with MSOS, we see that MSOS is more resistant towards unforeseen extensions of a language description. Adding new constructs to a MSOS description can involve adding a new field to the labels in the description, but this does not change the descriptions of the other constructs. One way of viewing AS is that it is MSOS with a fixed set of labels (An MSOS description of AN can be found in [62]). To solve the problem arising when we want to add an unusual construct to a language description, we could extend AN with a new action constant or action combinator by giving a MSOS definition of it and extending the set of labels in the MSOS description of AN; that way we would avoid breaking the modularity of the description. This of course assumes the existence of an MSOS description of the unusual construct.

An advantage of AS over MSOS and other semantic frameworks is the readability of AN. The intention of the English keywords in AN is that a semantic description should be readable by programmers not familiar with formal methods, and the ASD would then work as documentation of a language. Regrettably, not all the keywords are suggestive enough, for instance fail, maybe and else are related, but check and else (without the maybe) are not. Also keywords like furthermore and before do not suggest that the behaviour is related to bindings and what the behaviour is; but the context often helps.

The following example of MSOS rules describes a conditional expression:

$$\frac{\mathit{E1} \xrightarrow{x} \mathit{E1'}}{\mathsf{cond}(\mathit{E1},\ \mathit{E2},\ \mathit{E3}) \xrightarrow{x} \mathsf{cond}(\mathit{E1'},\ \mathit{E2},\ \mathit{E3})}$$

$$\mathsf{cond}(\mathsf{true},\ \mathit{E2},\ \mathit{E3}) \longrightarrow \mathit{E2}$$

$$\mathsf{cond}(\mathsf{false},\ \mathit{E2},\ \mathit{E3}) \longrightarrow \mathit{E3}$$

Comparing the rules with the action denoting the same expression in Subsection 2.3.3, we notice both that the rules are more complex and that the action is shorter. Another advantage is that previous work has shown that it is possible to construct tools, like programming environments, interpreters, and compiler generators, for working with AS. But when it comes to generating efficient compilers other (non semantic) frameworks which require a lower level description of a language would probably be more suitable, if we were willing to sacrifice the readability and simplicity of AS.

Compared to previous versions, the reduced size of the kernel in AN-2 is both an advantage and a disadvantage. It is useful in tools because instead of implementing, for instance, an action interpreter for a large notation we can implement it for the kernel and then reduce an action to kernel notation before using the interpreter. On the other hand it can be more difficult to infer the intention of an action when it has been reduced to kernel notation, which is an disadvantage in the tools that performs analysis of actions, for instance, the action compiler.

Another drawback is the rather small community using AS. In general the use of semantic frameworks for describing languages is not widespread, but among computer scientist it is our impression that the denotational and SOS frameworks are more common than AS. Hopefully AS will grow in popularity as the tools supporting it mature.

2.9 Restricted AN used in the dissertation

We have limited the subset of AN used in this dissertation in various ways. Most importantly we are not dealing with the actions for describing interactive processes. Including all of AN would make this project too big a task for us to handle, and since many languages do not include concurrency features, we did not feel that we restricted the expressiveness of AN too much by removing this part. In connection with inferring types and compiling actions we have restricted ourselves further, but this will be explained in later sections.

Chapter 3

ASF+SDF

Grammar is the logic of speech
— Richard C. Trench

The ASF+SDF formalism is actually two formalisms: SDF, the Syntax Definition Formalism which allows defining syntax using grammars in extended BNF form, and ASF, the Algebraic Specification Formalism for defining rewrite rules based on syntax defined in another formalism. ASF and SDF in combination can be used to define the syntax of a language together with a rewrite semantics based on the syntax. An ASF+SDF specification consists of a set of modules where each module consists of both an SDF and an ASF file.

3.1 SDF

SDF [4,27,87] is developed at CWI[1] and allows arbitrary, cycle-free, context-free grammars. Throughout this section we shall use the SDF modules *Expressions* and *containers/Table* in Figs. 3.1 and 3.2 as examples.

3.1.1 Declaring syntax

An SDF specification consists of a set of modules where each module defines parts of the syntax of the whole specification. A module can use syntactic sorts (nonterminals) defined in other modules by importing these modules. This is illustrated in line 2 in Fig. 3.1 where a module (*basic/Integers*) defining integers is imported. Modules have hierarchical names reflecting the directories of the file system where the modules reside. In the example in Fig. 3.2 the module *Table* is in the directory *containers*. The syntax defined by a module is the union of the syntax defined in the module and all imported modules. Forming the union of the syntax defined in different modules is possible because context-free languages are closed under union.

Not only can SDF modules import declarations, they can also export declarations, and this is indicated by the keyword **exports**. In the *Expressions* module lines 3 to 19 are exported. Declarations can also be local to a module

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

```
(1) module Expressions
(2) imports basic/Integers
(3) exports
(4)   context-free start-symbols Expression
(5)   sorts Expression Identifier
(6)   context-free syntax
(7)       Identifier | Integer → Expression
(8)       "if" Expression "then" Expression
                  "else" Expression → Expression
(9)       Expression "+" Expression → Expression {left}
(10)      Expression "*" Expression → Expression {left}
(11)      "(" {Expression ","}* ")" → Expression
(12)   context-free priorities
(13)      Expression "*" Expression → Expression {left}
(14)      >
(15)      Expression "+" Expression → Expression {left}
(16)   lexical syntax
(17)      [a-zA-Z]+ → Identifier
(18)   context-free restrictions
(19)      Identifier -/- [a-zA-Z]
(20) hiddens
(21)   variables "E"[0-9]? → Expression
```

Figure 3.1: SDF example

```
(1) module containers/Table[Key Value]
(2) imports
(3)     basic/Booleans
(4)     containers/List[Key]
(5)     containers/List[Value]
(6)     containers/List[<Key, Value>]
(7) exports
(8)   context-free syntax
(9)     List[[<Key, Value>]] -> Table[[Key, Value]]
 ...
```

Figure 3.2: SDF module example

which means that they are only used in the ASF module attached to the SDF module, and this is indicated by the keyword **hiddens** (lines 20 to 21).

The syntactic sorts defined in a module should be listed after the keyword

sorts as illustrated in line 5. Sorts can also be declared as start symbols which means that they can be the top sort of a parsed term (this is shown in line 4).

In SDF both lexical (see lines 16 to 17) and context-free syntax (see lines 6 to 11) can be declared using *productions*[2][3]. A production is essentially a production rule with the little twist that instead of the usual ::= operator it uses → with the order of the operands reversed (the sort being defined is on the right-hand side). The left hand side of a production can contain literals (like `"+"`), sorts (like *Exp*), choice (like '*Identifier* | *Integer*'), and regular expressions (like {*Exp* `","` }`*`, a comma separated sequence of *Exp*'s[4]). SDF also provides a built-in notation for tuples as illustrated in Fig. 3.2 where the tuple '`<`*Key*, *Value*`>`' occurs several times.

For use in ASF rewrite rules SDF allows variables ranging over parse trees of various kinds to be declared. In line 21 variables `E0`, `E1`, etc. are declared to range over *Expression*.

The module *containers/Table* in Fig. 3.2 taken from the ASF+SDF library that comes with the ASF+SDF Meta-Environment (see Section 3.3) illustrates the more advanced features of the module system. The module has two parameters *Key* and *Value* (see line 1) which can be used as syntactic sorts in the module. Modules importing this module should give values to the parameters. This is illustrated in lines 4 to 6 where the parameters are passed on in the imports of the module *containers/List*. Syntactic sorts in a parameterised module can also be parameterised as illustrated with the sort *Table* in line 9. A *Table* with formal parameters *Key* and *Value* contains lists of *Key* and *Value* pairs (lists are used to define tables). Parameterisation can be used to define generic data types such as the tables defined in the module *containers/Table* where something of sort '*Table*`[[`*Integers*, *String*`]]`' can be used to store terms of sort *String* using terms of sort *Integers* as keys.

3.1.2 Resolving ambiguities

Since no restrictions, as LL(k), LR(k), etc., are put on the class of grammars used in SDF, the grammars can be ambiguous. To resolve ambiguities [19], SDF lets the user define associativity after production rules (like {`left`} in lines 9 and 10 in Fig. 3.1) and priorities in the **context-free priorities** section (lines 12 to 15). When dealing with lexical syntax, the longest match is often preferred. This is ensured by **context-free restrictions** as shown in line 19 where it is defined that *Identifier* cannot be followed immediately by a character in the character class `[a-zA-Z]`.

[2] Actually the only differences between the two ways of declaring syntax is that whitespace is automatically allowed between the symbols on the left hand side of a *context-free syntax function*, and *lexical constructor functions* in ASF can only be used on lexical syntax

[3] In [14] the word *function* is used, but to avoid confusing it with semantic functions we use the word *production*

[4] '{*SORT SEP*}*' is short for '((*SORT SEP*)* *SORT*)?'

3.2 ASF

ASF [4, 15, 27] is a formalism for expressing rewrite rules. The rules are defined using conditional equations as illustrated here:

```
[if-true] eval(E1) == true
          ------------------------------------
          eval(if E1 then E2 else E3) = eval(E2)

[default-if-false] eval(if E1 then E2 else E3) = eval(E3)
```

In the example we describe a function *eval* that evaluates conditional expressions. The first equation starts with a tag `[if-true]` naming the equation. If the tag starts with `default` (as the second equation), 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 cases where *eval*(*E1*) can be rewritten to `true` (possibly by applying other rewrite rules in the specification) we can rewrite the conditional to *eval*(*E2*) (the evaluation of the expression in the left branch of the conditional). Notice that the equations can contain variables (in this case they are capitalised) ranging over syntax trees (in this case expressions).

The keyword **when** or the sign $\Longrightarrow$ can also be used to separate the conditions from the conclusion in conditional equations. ASF has a tiny syntax; an equation is just a tag, an equality, and a set of conditions where the left and right hand side have the same syntactic sort defined in the SDF part of the module. The conditions can be either matching (T `:=` T'), negative matching (T `!:=` T'), positive (T `==` T'), or negative (T `!=` T'). Only the left hand sides of the matching conditions may contain uninstantiated variables.

When reducing a term using the equations, the ASF evaluator starts by searching for an equation among the non-default equations where the left hand side matches the term to be reduced. If more than one equation matches, the one with the most specific left hand side is chosen. Then the conditions are evaluated using the variables already instantiated while instantiating more variables. If all of the conditions succeed, the result is computed maybe using variables. If a condition fails, the whole equation fails, and another equation is tried.

3.3 The ASF+SDF Meta-Environment

The development of ASF+SDF specifications is supported by an interactive integrated programming environment, the ASF+SDF Meta-Environment [14, 20]. This programming environment provides syntax directed editing facilities for both the SDF and ASF parts of modules as well as for terms. It also provides well-formedness checking of modules and visualisation of the import graph and parse trees. A parser and a parse table generator using the SDF part of the specification are essential parts of the environment together with an ASF interpreter that allows terms to be rewritten. The environment offers all kinds of refactoring operations at the specification level: renaming of modules,

copying of modules, etc. Furthermore a library of predefined primitive types and data structures, e.g., booleans, integers, strings, lists, sets, etc., is available. The library contains also a growing collection of grammars of programming and specification languages, e.g., Java, C, CASL, SDF itself, etc.

Recently an ASF debugger has been connected to the Environment which allows the stack and source code to be inspected during a stepwise evaluation of the rewrite rules.

The user interface of the ASF+SDF Meta-Environment is shown in Fig. 3.3. Modules defining the concrete syntax of Pico (a toy language) have been opened. In the left pane we see a tree-structured view of the modules, and the right pane shows the graph of import relations of the modules.

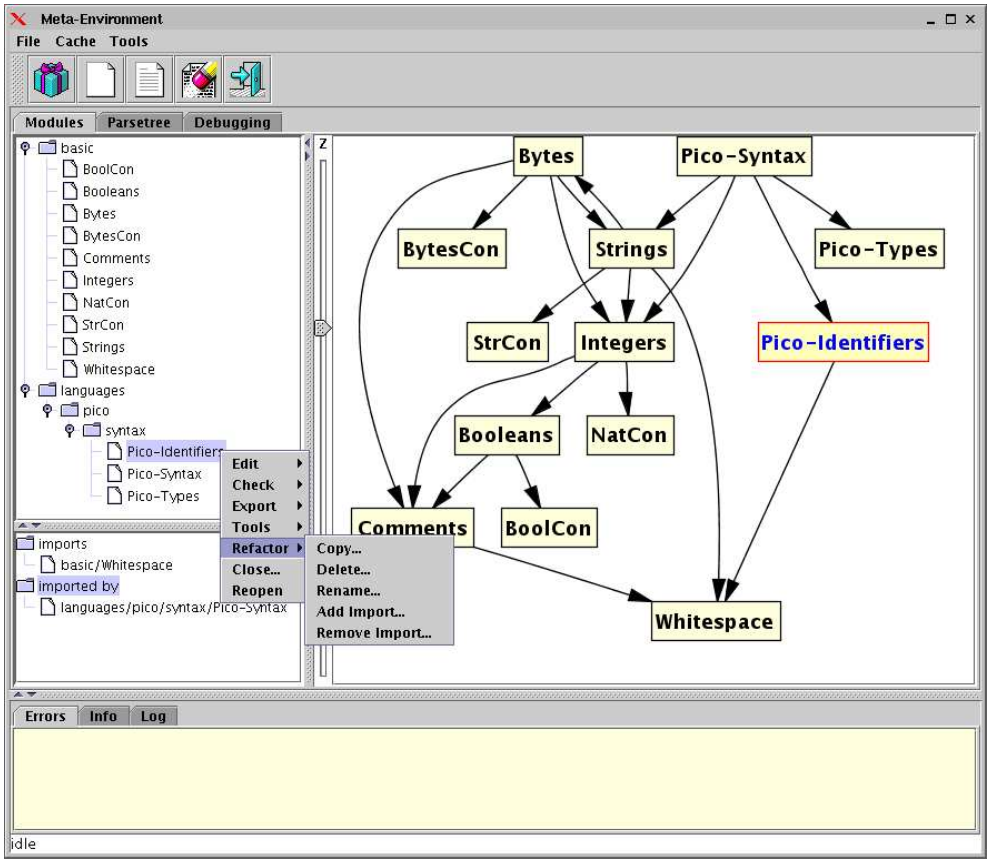

Figure 3.3: GUI of the ASF+SDF Meta-Environment.

3.4 Evaluation of ASF+SDF

We have used ASF+SDF as implementation language for most of the algorithms presented in this document. The language was originally designed as a language for describing the syntax of a language (using SDF) and various transformations on the language (using ASF). ASF+SDF can also be used as a

functional programming language, but as all languages it has both advantages and disadvantages.

3.4.1 Advantages

The SDF language allows arbitrary cycle-free context-free grammars. Instead of just giving an error message when a term is ambiguous it returns a forest of parse trees, and ASF can then be used to resolve the ambiguities by removing trees [16]. Furthermore it is also possible to parse a language with a context-sensitive grammar using ASF+SDF. In Chapter 5 we give an example of this.

ASF+SDF provides a high integration between the two languages because the terms used in the ASF equations are described in the SDF grammars. Furthermore both prefix constructor notation and the concrete syntax can be used in the ASF equations.

The conditional rewriting equations are easy to understand (see Section 2.18 in [14]) and essentially enough to express everything (ASF is Turing complete). The simplicity of the language should make it easy to learn. Built-in traversal functions [15] improve the expressiveness by letting the programmer ignore the trivial cases when traversing a parse tree. The library provides various tools, data types, and basic types and data operators that might be missing in ASF, like integers, booleans, etc. The type safe rewriting prevents many programmer bugs, especially when implementing translations between languages.

The built-in support for dividing a specification into parameterised modules makes it easy to write reusable and maintainable programs in ASF+SDF.

Finally an advanced environment for developing ASF+SDF specifications is available.

All in all ASF+SDF is a useful language for writing language translators and analysers.

3.4.2 Disadvantages

Most of the disadvantages we have noticed when using ASF+SDF are in connection with the ASF language.

One of the two main problems is the slow execution time of both the interpreted and the compiled ASF. The compiled version of our type inference algorithm (Chapter 9) has unacceptable running times on most large examples. The main problem seems to be the that there are no built-in integers and arrays. To improve the performance when rewriting a term, SDF provides the `memo` attribute which can be added to the production that defines the syntax of the term. Adding the `memo` attribute to a production tells the ASF interpreter to remember the result of rewriting terms derived from the production.

The other main problem has been the poor support for debugging. A trace of how the equations have been applied used to be the only help available for the programmer. The latest version of the ASF+SDF Meta-Environment remedies this problem by including an ASF debugger.

Related to debugging is the problem that has caused most bugs in our programs. The arbitrary grammars allowed in SDF can sometime make it

difficult to figure out how a term is parsed, and this can lead to unexpected behaviour when the left hand side of an equation is not parsed as expected and therefore does not match the term we expect it to rewrite.

The recent years has shown a lot of development in both the ASF and the SDF language, but unfortunately the new versions are not always backwards compatible, so old specifications need to be updated (a task the environment gives tool support for).

Regarding the expressiveness of ASF, we think the language can be improved by providing more control of which equations are tried. The negative conditions and default equations is enough to express the algorithms we have encountered in our work, but the it would save some superfluous negative conditions if the language had more than one `default` level. Conditional expressions in conditions could also alleviate the problem.

3.5 Alternatives

Considering the problems mentioned in the previous section it is worth investigating other languages as alternative implementation languages. We think that the biggest problems are related to the ASF language so it is natural to look at formalisms combining another rewrite formalism with SDF. Recent versions of ELAN [8,17] have used SDF for defining the signatures used in ELAN modules. ELAN modules offer rewrite rules, as ASF does, but it also lets the user define very advanced strategies for applying the rewrite rules. Comparisons between compiled ELAN and ASF specifications [11] has shown that ASF is slightly faster.

SDF supports interfaces to both Java and C so this could also have been a solution. Previous work on AS related tools has used Standard ML and C++ as implementation languages.

Chapter 4

ASDF

Now the whole world had one language
— Genesis 11:1

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

4.1 Using ASF+SDF to describe single constructs

We have previously used plain ASF+SDF for writing ASDs, as described by Doh and Mosses [29] and illustrated in Figs. 4.1 and 4.2. The two figures show two modules needed to define abstractions in a small lambda notation inspired language. An advantage of using ASF+SDF was that it allowed ASDs to be prototyped using the Meta-Environment. Furthermore other tools, like an action interpreter, action type checker, etc., could be connected to the Meta-Environment. However, using ASF+SDF for writing small modules describing single language constructs was not optimal, and this prompted the development of ASDF. The main problems with using ASF+SDF were related to the cumbersome notation:

- When using a syntactic sort, e.g., *Term*, in a production rule, the module introducing the syntactic sort had to be explicitly imported (see Fig. 4.2). Also modules describing AN had to be imported, since it was not part of the SDF language.
- The declaration of meta-variables ranging over sorts is somewhat tedious (see Fig. 4.1).
- ASF+SDF requires many keywords and they can be misleading, e.g., the signature of a semantic function is introduced by the words '**context-free syntax**'.

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

The ASDF modules corresponding to the modules in Fig. 4.1 and 4.2 is explained latter in this chapter (see Figs. 4.8 and 4.9).

```
module Term
  exports
    sorts Term
    context-free syntax
       "eval" Term → Action
    variables
       "T"[0-9]? → Term
       "T"[0-9]?"*" → Term*
       "T"[0-9]?"+" → Term+
```

Figure 4.1: Module *Term* in SDF

```
module Term/Abstract
  imports Term Data/Lambda Ide
  exports
    context-free syntax
       "abstract" "(" Ide "," Term ")"    → Term
       Lambda → Val

equations

[1] eval abstract (I, T) =
            give (lambda(closure(
              furthermore bind(the token I, the val)
              scope eval T)))
```

Figure 4.2: Module *Term/Abstract* in ASF+SDF

4.2 Definition of ASDF

In Figs. 4.3 and 4.4 the syntax of ASDF is defined using SDF.

Several syntactic sorts are not fully defined in the figure: *Sort* contains names of syntactic sorts (words starting with a capital letter), *Literal* contains quoted and unquoted literals (words starting with a small letter), *Section* contains SDF sections, *Symbol* contains among others *Sort* and *Literal*, *ActionType* contains types of actions, and *Action* contains actions.

A semantic description of a language consists of a collection of ASDF modules, together with a mapping from the concrete syntax used in the language to the abstract syntax described in the modules. Fig. 4.5 shows a small lambda calculus language, and the Modules 4.7 to 4.12 can be used to describe the constructs found in the language. The mapping from the concrete syntax of lambda language to the abstract syntax is described in Fig. 4.6, where T ranges over *Terms* and T^τ over the mapping of a term.

An ASDF module (*ASDF-Module*) consists of a name (after the keyword **module**) and a sequence of *ASDF-Sections* (usually there is at most three

```
module asdf
imports   sdf   an   types
exports
  context-free start-symbols     ASDF-Module
  context-free syntax
%% ASDF module
    "module" ModuleName ImpSection* ASDF-Section*
                                              → ASDF-Module

    "syntax" ASDF-Syntax* | "requires" ASDF-Requires* |
    "semantics" ASDF-Semantics*
                                              → ASDF-Section
```

Figure 4.3: SDF definition of ASDF syntax (part 1)

sections). The **syntax** section defines the abstract syntax of the construct. This is illustrated in Fig. 4.9 with the abstraction term constructor abstract which takes an identifier (*Ide*) and a term (*Term*) as argument. When writing production rules the separator '::=' is used, instead of the '→' found in SDF.

The **requires** section is used for introducing types, operators, and variables used in the **semantics** section. This is illustrated in Fig. 4.10, where the sort *Val* is extended with the sort *Lambda*, such that actions can produce lambda abstractions when evaluated. The same module illustrates declaration of constants; in this case the constant error, a value used as exception when a term cannot be applied. The syntax for declaring variables is illustrated in Fig. 4.8, where '*T* : *Term*' declares the variable *T* to range over the syntactic sort *Term*. When declaring the variable *X* to range over a sort *S* the variables *Xn*, *X**, and *X+*, where *n* is a non-negative integer, are automatically declared to range over the sorts *S*, S^*, and S^+. The use of these variables is illustrated in Fig. 4.10.

Fig. 4.12 illustrates how types and operators are introduced. The declaration '*Lambda* ::= lambda(act: *Action* & using val & giving val)' results in the type lambda and the data operators lambda and act becoming available in actions, so that we can write actions such as 'give the lambda' and 'give act(...)'. The operator lambda is a data constructor, and act selects the action component of such data. Notice also that every declaration of a data type, like *Lambda*, causes lambda to be declared as a type for use in AN. As a convention the types in AN use lower case start characters.

The semantic function, mapping the abstract syntax construct introduced in the **syntax** section to an action, is defined, using an equation, in the **semantics** section. In the equation, terms from AN and imported modules can be used. For instance, in Fig. 4.10 the semantic function contains action combinators and constants, together with the type lambda, declared in the imported module *Data/Lambda*. It is possible to define the function using more than one equation, this is illustrated in Module 14 on page 56. The **semantics** section can also contain the signature of a semantic function, as we see in Fig. 4.8. It is required that the signature of a function, used in a module, is defined in the same module

```
%% Syntax section

    Sort "::=" ASDF-Syntax-Rhs          → ASDF-Syntax

    Sort | Literal | Literal "(" {Symbol ","}* ")" |
    ASDF-Syntax-Rhs "|" ASDF-Syntax-Rhs    → ASDF-Syntax-Rhs

%% Requires section

    VarLexPrefix ":" Sort | Literal ":" Sort |
    Sort "::=" ASDF-Requires-Rhs              → ASDF-Requires

    Sort |
    Literal "(" {(Literal ":" (Sort | ActionType)) ","}* ")" |
    ASDF-Requires-Rhs "|" ASDF-Requires-Rhs
                                              → ASDF-Requires-Rhs

%% Semantics section

    ASDF-Equation | ASDF-Signature              → ASDF-Semantics

    "[" UQLiteral? "]" Literal Constructor "=" Action
                                              → ASDF-Equation

    Literal ":" Sort "→" ActionType         → ASDF-Signature

    VarLex | Literal | Literal "(" {VarLex ","}+ ")" |
    Literal "(" VarLex* ")"                   → Constructor

    "(" Symbol ")"              → Symbol

    Literal Constructor         → Action

  lexical syntax

    [A-Z]+[0-9\']*[\+\*]?       → VarLex
    [A-Z]+                      → VarLexPrefix
```

Figure 4.4: SDF definition of ASDF syntax (part 2)

```
Term  ::=  Ide | Term Term | λ Ide . Term
```

Figure 4.5: Small lambda language

or an imported module. The notation used in a semantic equation (besides

$$
\begin{array}{lcl}
I & \rightarrow & \texttt{ide}(I) \\
T_1 T_2 & \rightarrow & \texttt{apply}(T_1^\tau, T_2^\tau) \\
\lambda I.T & \rightarrow & \texttt{abstract}(I, T^\tau)
\end{array}
$$

Figure 4.6: Mapping lambda language to abstract syntax

module *Calculus*

imports

Term/Abstract
Term/Apply
Term/Ide

Figure 4.7: Module *Calculus*

module *Term*

requires

T : *Term*

semantics

eval : *Term* → *Action* & using () & giving val

Figure 4.8: Module *Term*

module *Term/Abstract*

syntax

Term := abstract(*Ide*, *Term*)

requires

Val ::= *Lambda*

semantics

[1] *eval* abstract(*I*, *T*) =
give (lambda(closure(
furthermore bind(*I*, the val)
scope eval *T*)))

Figure 4.9: Module *Term/Abstract*

module *Term/Apply*

syntax

Term := apply(*Term*, *Term*)

requires

Val ::= *Lambda*

error : *Val*

semantics

[1] *eval* apply(*T1*, *T2*) =
eval T1 and *eval T2* then
maybe apply (act(the lambda), the val)
else throw error

Figure 4.10: Module *Term/Apply*

module *Term/Ide*

syntax

Term := ide(*Ide*)

semantics

[1] *eval* ide(*I*) = give the val bound-to *I*

Figure 4.11: Module *Term/Ide*

module *Data/Lambda*

requires

Lambda ::= lambda(act: *Action* & using val & giving val)

Figure 4.12: Module *Data/Lambda*

AN) is defined in the **syntax** and **requires** sections of the module and imported modules. Therefore parsing a module must be done in two steps, where the first step builds a parsetable based on the **syntax** and **requires** sections. More about this in Chapter 6.

Syntactic sorts used in the **syntax** section result in implicit imports, so for instance in Fig. 4.9 the modules *Term* (Fig. 4.8) and *Ide* (not shown) are automatically imported. Implicit imports are also generated from the sorts

used in the **requires** section, with the difference that only syntactic sorts used on the right hand side of the production results in imports, and the imported modules always start with *Data/*, for instance Fig. 4.10 imports *Data/Lambda* (Fig. 4.12). The automatically imported modules, like *Term* or *Data/Lambda*, may provide further sorts than those that caused their importation.

ASDF also allows explicit imports. This is mostly used in the top module that imports all the modules used to describe a language (see Fig. 4.7). The import-relation between the modules can be seen in Fig. 4.13, where a module A imports a module B if there is an arrow from A to B.

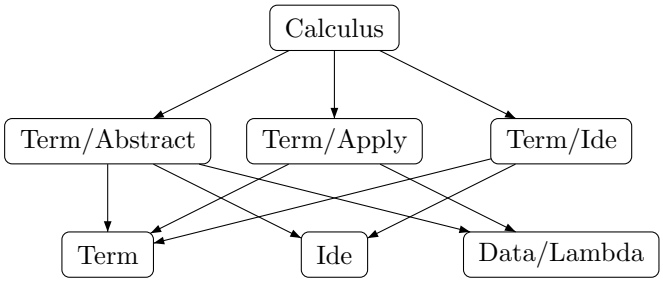

Figure 4.13: Import graph for Lambda language

ASDF only supports prefix constructor abstract syntax instead of concrete syntax when describing language constructs. The advantage of using language independent prefix constructors for abstract syntax is greater reusability. For instance, a description of the if-then-else expression from Standard ML might be reused for describing the '?:' expression in Java, since they have the same compositional structure and intended interpretation even though their concrete syntax differs.

4.3 Future work

The ASDF formalism is still at the experimental level and has only been tested on the Core ML example. Parametrised modules as found in SDF might result in greater reusability. Extending ASDF with support for user defined action abbreviations can help simplify the semantic equations. This would require extending the syntax used in the **requires** section so that the user can define the syntax of a new action combinator. Also the syntax of the semantic equations should be extended so that the expansion of the abbreviation can be defined.

Allowing conditional equations, as found in ASF, in the **semantics** section would extend the expressiveness of ASDF. It is not clear whether it is needed since the construct described in an ASDF module should map easily to an action, and if conditional equations are needed, it might indicate that the construct is too complex and should be split into more constructs to improve the reusability of the constructs.

Chapter 5

Core ML example

A wisely chosen illustration is almost essential to fasten the truth upon the ordinary mind, and no teacher can afford to neglect this part of his preparation.
— Howard Crosby

This chapter presents a constructive action semantics of Core ML. Doh and Mosses [29] 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 [4, 27]; in contrast, we use ASDF.

In [29] 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.

The contents of this chapter is based on *Constructive action semantics for Core ML* [41].

5.1 ML syntax

ML [51, 85] 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 grammar of the whole Core ML syntax, which is consistent with the grammar found in *The Definition of Standard ML* [51]. In the next sections we will use the constructs introduced in this

section as examples when giving a semantics for Core ML. Readers already familiar with ML might prefer to take a quick look at Fig. 5.1 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.

Fig. 5.1 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*.

```
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
```

Figure 5.1: ML Grammar

5.1.1 Expressions

ML does not 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 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 data types 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 do not exist in ML: '`(`*EXP*`)`' is just used for grouping expressions.

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

5.1.2 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 Appendix A.2.

5.1.3 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 Subsection 5.1.1 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 IDE0  =  IDE1 of TYP1
               |  ...
               |  IDEn of TYPn
```

where IDE_0 is the name of the new datatype and IDE_1, ..., IDE_n are the names of the data constructors. The types TYP_1, ..., 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 Appendix A.3.

5.1.4 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 Appendix A.4 for the full specification of the syntax of types in Core ML.

5.1.5 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.

Another problem related to identifiers is the values they are bound to. In

```
let
  datatype boolean = TRUE | FALSE;
in
  (fn TRUE => 1 | _ => 0) FALSE
end
```

the behaviour of the anonymous function depends on the fact that the identifier '`TRUE`' is bound as a data constructor. This means that an identifier should either be regarded as a constant or a normal identifier which can be bound to a value when it occurs in a pattern. Distinguishing between these two kinds of identifiers can either be done in the semantics of identifiers (see Subsection 5.2.1) or when constructing a BAS term (more about this in Section 5.2). Using semantics to distinguish is described in Module 4 on page 51 and Module 11 on page 55. Distinguishing when constructing the BAS term requires more BAS constructs to represent different kinds of identifier use and an extra traversal of the parse tree to determine the different uses (more about this in Section 9.7).

5.2 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 mapping is described in Appendix B.

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 here 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 does not produce any data when evaluated. 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 Fig. 5.1. 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*.

Fig. 5.2 shows an example of how ML is mapped to BAS.

5.2.1 Expressions

The function *exp2bas* is fully described in Appendix B.1. In this section we shall see some examples from this description. The examples are listed in Fig. 5.3.

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

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.

```
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(exn))
     rec(bind-val(var(fac),
       app-seq(abs(var(f), abs(var(id0),
          app-seq(val(f), val(id0)))),
        alt-seq(
         abs(0, 1)
         abs(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))
   )
```

Figure 5.2: Mapping ML to BAS

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. 5.2.3.

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

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*($E_1^\top$, $E_2^\top$, $E_3^\top$)
`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*($E_1^\top$ ... $E_n^\top$) , $n \geq 2$
`(`E`)`	$\rightarrow$	$E^\top$
`[`E_1`, ..., `E_n`]`	$\rightarrow$	*app*(*list*, *tuple-seq*($E_1^\top$, ..., $E_n^\top$))

Figure 5.3: ML expressions to BAS mapping

a statement from an expression by discarding the value.[1]

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 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*(::), ...

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

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* [51, Appendix D]. It is also possible to give ASDs of these operations, but we shall omit the details here.

5.2.2 Patterns

The function used for mapping ML patterns to BAS parameters is named *pat2bas*. The complete definition of it can be found in Appendix B.2. In this subsection we will only elaborate on the subset of the mapping displayed in Fig. 5.4.

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

Figure 5.4: 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 Subsection 5.2.1.

List patterns (Fig. 5.4) are very similar to list expressions (Fig. 5.3) 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 does not matter, and therefore none of the BAS constructs used in this subsection insist on sequential evaluation.

5.2.3 Declarations

Appendix B.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 Fig. 5.5

`val` P `=` E	$\rightarrow$	*bind-val*($P^\top$, $E^\top$)
`fun` I P `=` 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*(*var*(I_1), *new-cons*(I))
`|` ...	$\rightarrow$	...
`|` I_n `of` T_n		*bind-val*(*var*(I_n), *new-cons*(I)))
`exception` I `of` T	$\rightarrow$	*bind-val*(*var*(I), *new-cons*(*exn*))

Figure 5.5: 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 Appendix B.3. The mapping of '`fun` I P `=` E' can be found in Fig. 5.5, 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*(I)) 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*(*exn*)). We do not see any reason to distinguish between exception constructors and data constructors.

5.3 Action Semantics for Basic Abstract Syntax

In this section we will describe the semantics of selected BAS constructs using AS. The description is written in ASDF as presented in Chapter 4, with the only difference that the keyword **module** has a number appended for easier reference. The rest of the constructs used in the description of Core ML can be found in Appendix C. Fig. 5.6 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 (val(the cons bound-to the token x) = the val)
       then give (no-bindings))
     else
      (give (the cons bound-to the token x) then fail
       and-catch bind(the token x, the val))))
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))
```

Figure 5.6: Mapping BAS to AS

To combine all the modules needed to describe Core ML, we have the module *CoreML*:

Module 1 *CoreML*

imports

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
Exp/New-Cons

Stm/Exp
Stm/While

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

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

The module does not import all modules mentioned in the following sections because some of them are implicitly imported from other modules.

5.3.1 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 2 *Exp*

requires

E : *Exp*

semantics evaluate: *Exp* → *Action* & using data & giving val

The module defines the semantic function evaluate and variables starting with *E* to range over expressions. The signature of evaluate expresses that the function takes an *Exp* and returns an action that can be given all kinds of data and

produces values. The information about what kind of action it returns is used by the semantic function type checker presented in Chapter 7. We could have been more restrictive about the data given to the action by adding 'using ()' to the signature (expressions does not refer to any given data), but then we would have to insert 'then skip' before applications of evaluate in some of the semantic equations for them to type check.

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

Module 3 *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 give the val *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(T)* construct:

Module 4 *Exp/Val-Id-Const*

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

requires

Val **::=** *Cons*

Bindable **::=** *Val*

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

Data constructors belongs to the sort *Cons*. The data operation val is used to construct a value from the data constructor. In the action the yielder 'the cons bound-to the token *T*' is used, which looks up the data constructor bound to *T* in the current bindings. In AN we use tokens instead of identifiers to bind values, but we shall define *Token* to include *Ide*. It is the data operation 'the cons', which ensures that *T* is bound to a data constructor. The action constant give applies a yielder to the given data. If *T* was bound to something not of sort *cons* the give action would terminate exceptionally, in which case maybe and else make sure that an alternative is tried. The alternative is to try to give the value bound to *T*, 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 5 *Exp/App-Seq*

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

requires

Val **::=** *Func* | *ConsData* | *Cons*

func-no-apply : *Val*

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

In the **requires** section we make sure that functions, constructed data, data constructors, 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. An application in ML can either apply a data constructor to a value and result in a constructed value, or it can apply a function to a value and result in a value. The two actions on each side of the first else combinator describe the two different behaviours. The first action uses the built-in data operator consdata to construct data, containing information from the two expressions. The reason consdata is built-in, and not just defined in the **requires** section, is its connection with the invert data operator presented in Module 12 on page 55; consdata constructs data with a data type constructor and invert deconstructs the same data if it is given the right data type constructor. If the first expression is not a data constructor the action fails, and the alternative is tried. Again we see the use of the data operation 'the τ', where τ is a type; 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 does not fail, the result of the whole action is just the result of the application.

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 6 *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; 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 *E2* and that of *E3*.

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

Module 7 *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.[2] The 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 AN: actions as data, and scopes of bindings.

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*)))

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

[2]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.

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 does not introduce any new AN, and instead we will take a look at another module concerned with exceptions.

Module 9 *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.

5.3.2 Statements

Although ML does not 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 10 *Stm/While*

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

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

semantics execute while(*E*, *S*) =
unfolding (evaluate *E* then
maybe check (not(the boolean))
then skip
else (execute *S* then unfold))

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*.

5.3.3 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 11 *Par/Val-Or-Var*

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

requires

Val **::=** *Cons*

Bindable **::=** *Val*

semantics match val-or-var(*T*) =
(maybe check (val(the cons bound-to the token *T*) = the val)
then give no-bindings)
else
(give (the cons bound-to the token *T*) then fail
and-catch bind(the token *T*, the val))

Data constructors belongs to the sort *Cons*, but we can get a value from data constructors using the data selector val. The construct *val-or-var(T)* is mapped to an action that first checks whether *T* is bound to a data constructor and then compares the given value to the constructor. If they match, the result is the empty set of bindings. If they do not match or *T* is not bound to a data constructor, the alternative is to check if *T* is bound to a data constructor and then fail, or bind *T* to the given value. Since a value is bound to a token in the semantic equation, the **requires** section must declare values to be bindable.

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

Module 12 *Par/App*

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

requires

Val **::=** *Func* | *ConsData* | *Cons*

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.

5.3.4 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 13 *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 14 *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 the rest without using the bindings from the first declaration. 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 15 *Dec/Rec*

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

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

AN contains an action combinator that does exactly this, 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 16 *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 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.

5.3.5 Data

Many of the modules presented in the previous sections implicitly import modules from the *Data* directory.

Some of the modules are empty like

Module 17 *Data/Bindings*

because bindings are already a part of AS and therefore a part of ASDF, but the way ASDF implicitly import modules require that the modules exists. Others like

Module 18 *Data/Func*

requires

Func **::=** func(action: *Action* & using val & giving val)
| datacons(token: *Token*, tag: *Cell*)

describe data, types, data constructors, and data selectors used in the modules importing them. This module describes two ways of constructing data of type *Func*: Either the data constructor func is applied to an action (the type of the action is explained in Chapter 7), or the constructor datacons is applied to both a *Token* and a memory cell. This describes that ML functions can either be ordinary functions or data constructors.

5.4 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.

5.4.1 The syntax of Core ML

We have chosen to start from the syntax for Core ML given in *The Definition* [51, Appendix B], reformulated as a grammar in SDF as shown in Appendix 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 formal semantic descriptions.

5.4.2 Mapping from Core ML to BAS

The complete mapping is specified in Appendix 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' declarations given in *The Definition* [51, Appendix A] somewhat clumsy, so we use a simpler translation, totally avoiding the need for creating 'fresh' variable identifiers. The basic idea is illustrated in Fig. 5.7.

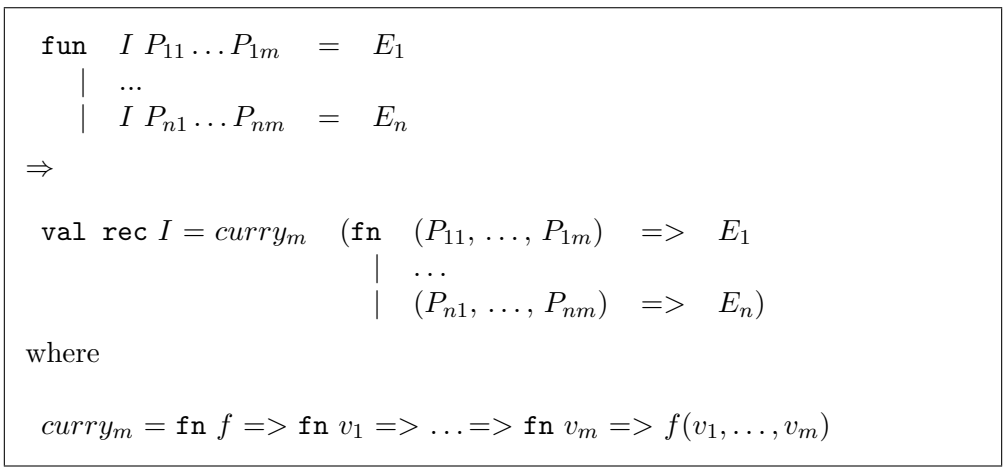

Figure 5.7: Expansion of `fun` declarations

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.

5.4.3 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 subparameter, 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 sublanguages 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.

5.5 Related work

Watt [90] reported on a previous case study concerning the use of AS 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* [51]. 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 AS 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 Fig. 5.8 we have illustrated the parts of Watt's 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 Watt's description does not have the same degree of modularity as ours.

It is difficult to compare the size of Watt's ML description [90] and our description due to the structural differences. Watt's 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.

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 19 *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*

Figure 5.8: Watt's and our description of the conditional expression

Part II

Tools

Chapter 6

The Action Environment

Give us the tools and we will finish the job.
— Sir Winston Churchill

The contents of this chapter is taken from *An action environment* [12], and describes the Action Environment, an environment for working with the formalisms described in part I.

6.1 Features

The Action Environment supports working with ASF+SDF and ASDF simultaneously, with the restriction that ASF+SDF modules can import ASDF modules, but not the other way round. If there is a name conflict, i.e., an ASF+SDF module and an ASDF module with the same name, it is solved by using the module with the same type as the module importing the problematic module. Being built on top of the ASF+SDF Meta-Environment, the Action Environment inherits most of its features (described in Section 3.3).

A screen dump of the Action Environment can be seen in Fig. 6.1. On the surface the differences between the Meta-Environment and the Action Environment seem negligible. Because a module in the module graph can be either an ASDF or an ASF+SDF module, different pop-up menus will appear over modules of different type. Not all features available for ASF+SDF are available for ASDF because they have not yet been implemented (e.g., changing module name and imports). When editing an ASDF module one notices more differences, since the syntax directed editor now uses an ASDF grammar for parsing. Furthermore, the grammar defined in a module (and in the modules it imports) is used when parsing the semantic equations in a module (remember that the equations and the rest of the ASDF module is in the same file, and not in two files as in ASF+SDF). This has two advantages: It gives a better syntactic check of the semantic equations, and it allows the syntax directed structure editor to display the right sorts for the tokens in the semantic equations. As in the Meta-Environment, it is possible to employ the given language specification for parsing and rewriting terms over the language. Due to the way we implemented the Action Environment, everything concerning terms works as in the Meta-Environment.

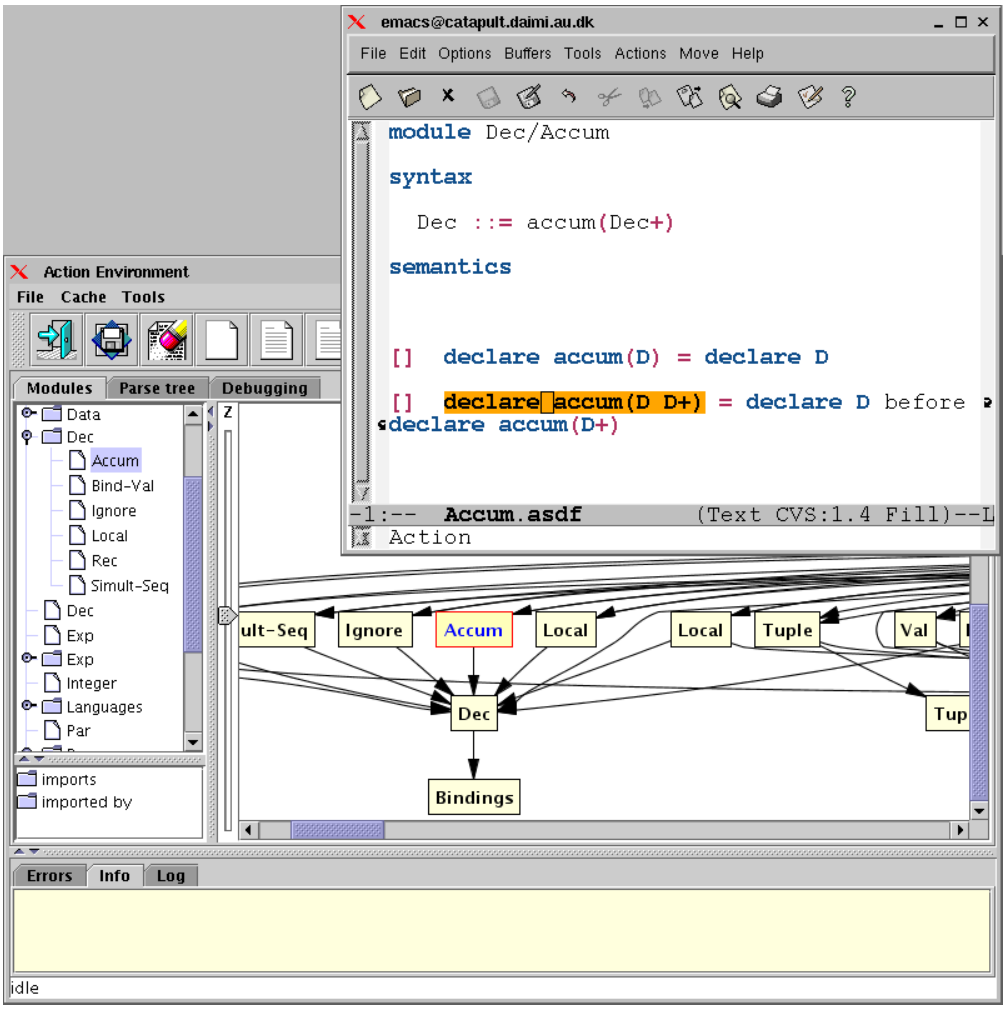

Figure 6.1: The Action Environment

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

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

Different external tools have been integrated into the Action Environment. A type checker for action semantic functions gives us a better check of the

well-formedness of the ASDF modules and thereby the correctness of the ASD of the language. An action interpreter allows us to interpret programs written in the language we are describing. All in all, the Action Environment should provide a particularly useful environment for developing semantic descriptions and documenting the design of programming languages.

6.1.1 Tools

Both the ASDF type checker, described in Chapter 7, and the action interpreter, described in Chapter 8, can be invoked from the Action Environment.

When type checking the semantic equations in a module, the user should provide signatures for the semantic function they define and the semantic functions they employ, and the type checker then checks that the semantic equations conform to the signature of the function they define.

Type checking will either result in an error message indicating what might be wrong in the action, or a message saying that the equations type checked without problems. Because this is a soft type check, the purpose is not to guarantee that the actions resulting from applying the semantic functions are well formed. Instead the purpose is to warn the language describer against possible problems in the specification.

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

6.2 Implementation

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

6.2.1 ASF+SDF Meta-Environment architecture

The Meta-Environment has a layered architecture as displayed in Fig. 6.2. In this section we will discuss each of these layers in more detail. The first step towards a layered design of the ASF+SDF Meta-Environment is discussed in [18]. That paper discusses how ASF can be replaced by another rewriting formalism. This development has been taken a step further, resulting in the architecture discussed here.

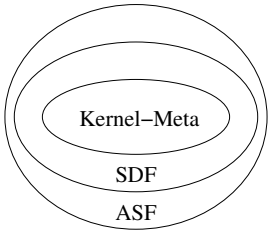

Figure 6.2: The layered architecture of the ASF+SDF Meta-Environment

Kernel layer

The kernel of the Meta-Environment is completely language-independent. It consists of the software coordination architecture, the ToolBus [5], which takes care of all the communication between the components that make up the Meta-Environment. The ToolBus allows a full separation of coordination and computation: it is a programmable software bus where the coordination between the components is formally described using a Process Algebra based formalism. The computation is performed within the connected components, which can be implemented in any programming language. The exchange of data between the components is based on a representation format, ATerms [9], specially designed for representing tree-like data structures. This formalism provides maximal subterm sharing and efficient linearisation operations.

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

SDF layer

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

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

Table 6.1: The Meta-Environment hooks: hooks that parameterise the GUI

ASF layer

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

Implementation

Fig. 6.3 shows an abstraction of the kernel Meta-Environment with each of the extensions described above. In this section we will briefly describe how we achieve these extensions in a flexible way.

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

We identify messages that are sent from the kernel of the Meta-Environment to the extensions as so-called *hooks*. The SDF layer can and will introduce new hooks for the next layers. Each instance of the environment should *at least* implement a receiver for each of these hooks. Implementing these hooks involves writing small pieces of ToolBus specifications. Table 6.1 shows a few kernel hooks. They are all related to the GUI and editors. The dashed arrows in the Fig. 6.3 between the kernel layer and the ASF or SDF layer denote the hooks and the service requests.

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

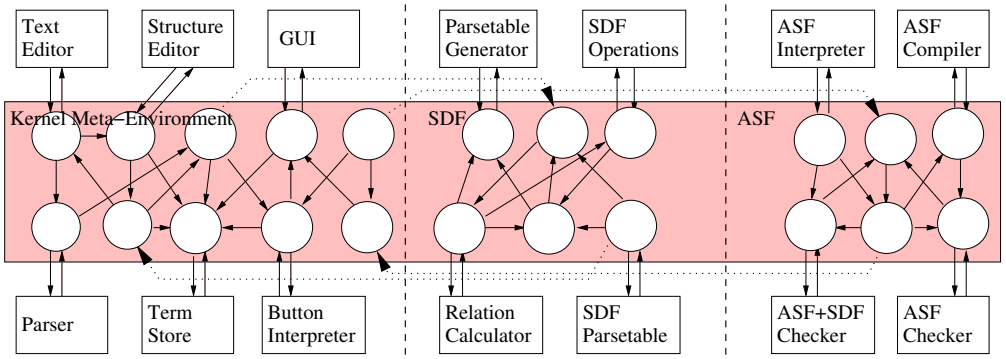

Figure 6.3: The layered implementation of the ASF+SDF Meta-Environment

6.2.2 The Action Environment

In the Action Environment the layered design of the Meta-Environment is extended with an extra layer, the ASDF layer, illustrated in Fig. 6.4. Notice that we do not replace any parts of the ASF+SDF Meta-Environment; we just extend it with an extra layer on the top. In [18] it is described how an environment for another rewriting formalism is implemented by replacing the ASF layer with a layer for the new formalism. This approach is not possible for us because the Action Environment should still support ASF+SDF modules. Another way of viewing the ASDF layer is as an ASDF interface to the ASF+SDF Meta-Environment.

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

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

Mapping ASDF to ASF+SDF

The Action Environment contains two mappings of ASDF to ASF+SDF. The result of one mapping is used for parsing and rewriting terms. By mapping every ASDF module to an ASF+SDF module we get the same effect, with respect to working with terms, as if we had opened the generated ASF+SDF modules in the Meta-Environment, so editing of terms is independent of the

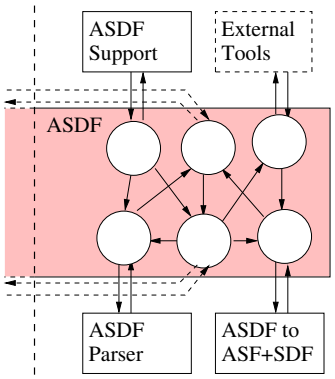

Figure 6.4: The ASDF layer

ASDF layer. The result of the other mapping is used for the second parse of the ASDF module itself (the parse that checks the semantic equations using the notation introduced in the same module and modules imported from it).

We shall use some of the modules in Section 5.3 as examples in this section. The ASDF module's name declaration together with its import section (if any) can be copied verbatim into the ASF+SDF module as illustrated in Fig. 6.5. Together with the explicit imports from the ASDF module, the generated ASF+SDF also contains imports of modules describing AN (the module *AN*) and layout characters (the module *Layout*). Implicit imports, as explained in Section 5.3, are translated to explicit imports, e.g., Module 5 on page 52 uses the sorts *Exp* and *Func*, and the SDF generated from this module imports the modules *Exp* and *Data/Func* (Fig. 6.7).

module *SmallML*

imports

Exp/Val
Exp/Val-Id-Const
Exp/App-Seq
Exp/Tuple-Seq
Exp/Cond

...

imports *AN* *Layout*

Figure 6.5: SDF generated from Module 1 on page 50

The rest of an ASDF module is translated into ASF equations and SDF sections declaring start symbols, sorts, lexical and context-free syntax productions, and variables. The sort declaring sections ensure that all sorts occurring on the right hand side of the arrow in a syntactic function are declared. Examples of this can be seen in Fig. 6.6 where the sorts *Exp*, *Type*, and *Action* are declared.

```
module Exp

imports
  AN  Layout  Data/Val

exports
  sorts Exp

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

  sorts Action

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

Figure 6.6: SDF generated from Module 2 on page 50

The sorts which are also declared to be context-free start symbols in ASF+SDF can be used as the top sort in a parse tree for a term. All sorts defined in the syntax section of an ASDF module are declared to be start-symbols (see Fig. 6.7).

A production of the type '*Sort* ::= *Symbols*' in the **syntax** and **requires** sections is mapped into a context-free syntax section containing a function '*Symbols* → *Sort*', as shown in Fig. 6.7. The productions in **requires** sections also result in declaration of types for use in AN, e.g., in Fig. 6.7 the production '*Val* ::= *Func* | *ConsData* | *Cons*' is translated to SDF that declares `val`, `func`, `consdata`, and `cons` to be types for use in action notation.

Module 2 on page 50 declares variables with the prefix `E`, and this is translated to the variables section shown in Fig. 6.6. Here regular expressions over character-sets and strings are used to define variables ranging over *Exp*, *Exp**, and *Exp+*. The variables are used in the ASF generated from **semantics** sections, as shown in Fig. 6.8. In the semantics section it is only the equations, and not the signatures, that are translated to ASF. The signatures are translated to a syntactic function as shown at the bottom of Fig. 6.7.

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

Integration of external tools

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

```
module Exp/App-Seq

imports
  AN  Layout
  Data/Func  Exp

exports
  context-free start-symbols Exp

  sorts Exp

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

  sorts Val Type

  context-free syntax
    Func              -> Val
    ConsData          -> Val
    Cons              -> Val


  lexical syntax
    "val"      -> Type
    "func"     -> Type
    "consdata" -> Type
    "cons"     -> Type

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

Figure 6.7: SDF generated from Module 5 on page 52

```
equations

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

Figure 6.8: ASF generated from Module 5 on page 52

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

Fig. 6.9 shows the definition of the menu item that starts the ASDF type checker written in button language. In the Meta-Environment anything the user can click is referred to as a button, hence also a menu item. The first line defines where the button should occur, and in this case it only occurs

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

Figure 6.9: Definition of type checking menu item

in ASDF editors. The second line describe how it should occur, and in this case it occurs as a menu item named "Type check" under the menu "Actions". The rest of the lines define the button's behaviour, using a special stack-based script language. The command `push-active-module` pushes the name of the module in the editor on the stack, before the command `prompt-for-file` asks the user for an ASDF file containing extra type information. Using the stack, the name of the file is passed to the next command (`split-file-name`) which splits the file name into directory, name, and extension. Finally the command `type-check-asdf` calls the ToolBus interface to the type checker.

6.3 Related work

An enormous amount of work has been performed in the field of defining the syntax and semantics of programming languages and systems supporting the development of such language definitions. We refer to Heering and Klint [37] for a fairly complete and up-to-date overview.

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

The Gem-Mex system [3] allows description of languages using a collection of Montages, a formalism based on Abstract State Machines. The idea of describing single language constructs in separate modules is encouraged by Gem-Mex, but due to the lacking modularity of the syntax formalism used (the semantic descriptions of individual constructs are based on concrete syntax, and the collected syntax has to be LALR(1)) a Montage is not often reusable in descriptions of different languages.

The Abaco system [72] is an AS tool for students and programming language designers. The main components of Abaco are an algebraic specification compiler, specification editors, action libraries, action editors, and a GUI. Furthermore, it offers a help system, an action debugger and facilities to export specifications to readable output. The main component is the algebraic specification compiler, which provides syntax checking of specifications and interpretation. The Abaco system and the Action Environment have a strong resemblance, but the Action Environment offers more flexibility in adding external components by means of openness of the underlying architecture.

The ASD toolset [26] supported the creation, editing, checking, and use

of ASDs. This toolset had a very strong relation with an older version of ASF+SDF, and its implementation has become obsolete.

6.4 Conclusion and future work

In this chapter we described the Action Environment, a new environment supporting the use of ASDF and ASF+SDF, and explained how it is implemented on top of the ASF+SDF Meta-Environment. The Action Environment has been tested on the Core ML example presented in Chapter 5.

Plans for future work on the Action Environment include:

- Testing the environment on more examples;
- Adding refactoring features for ASDF modules, such as renaming modules, adding and removing imports, deleting modules etc.;
- Validity check of ASDF modules should be improved: besides the syntax check there should also be checks of whether the right-hand sides of equations use variables not occurring in the left-hand side; and
- Adding functionality to merge two modules if they define the same construct, as described in [29].

Chapter 7

Type checking semantic functions

To err is human, but to really foul things up requires a computer.
— Dan Rather

This chapter is about type checking semantic functions as they occur in the ASDF formalism. The contents of this chapter is based on *Type checking semantic functions in ASDF* [39].

7.1 Type checking

Type checking in connection with AS can be done at two levels: Either semantic functions in an ASD are type checked to reveal mistakes made by the language describer, or the actions resulting from applying the semantic functions to a program are type checked to reveal errors in the program and to support code generation (if a type has been inferred for all subactions). The topic of this chapter is the former.

Type checking a semantic function is done one module at a time by type checking the semantic equations in a module defining the semantic function's behaviour on a single construct. By type checking semantic equations we mean checking that the equation conforms to the signature of the semantic function it defines. This of course requires information about all the user defined semantic functions, types, data, and data operators used in a semantic equation, and it requires that information can be collected from the module containing the equation and the modules it imports. If the action in a semantic equation contains type errors or its type is not a subtype of the expected type (according to the semantic function signature), we can report an error. We shall say that an action has a type error if one of its subactions terminates abruptly because it is given a type of data it did not expect. An example of an action containing a type error is the action 'result 5 then close'. This action is flawed because close expects an action, but receives an integer. If execute has the signature 'execute : *Stm* → Action & using data & giving ()', the semantic equation "execute new(*E*) = evaluate *E* then create" will also result in a type error because the signature does not allow that actions resulting from execute give memory cells.

Type checking of semantic equations is obstructed by the fact that the action on the right hand side appears out of the context it will appear in when

the semantic equation is used to map a complete program. A conservative type checker would reject many semantic equations because of the lack of information about the context. It is worth considering whether we can use quantified types or principal typings [91] to solve the problem with the missing context. We could use quantified types where we quantify over the tokens bound in the context, or we could use principal typings to define type judgements that describe the context, but the main problem is that type checking is done before token values are known (they will not be known until the semantic functions are applied to concrete programs), so we can never instantiate the quantified types or check the type judgements. Therefore quantified types or principal typings would not help us. We have chosen to develop a soft type checker that approves many actions but still warns the user against the most obvious mistakes.

The purpose of the type checker is to type check semantic functions in ASDF modules. Because of the modularity of ASDF descriptions we also want the type checker to be modular as explained in Section 7.5.

An implementation of the type checker, integrated into the Action Environment, has been used to type check the Core ML example.

7.2 Related work

Type checking (or type inference) of AN has been a research area since the beginning of the 1990's where Even and Schmidt [32] showed how to infer types for actions using unification on record types. Their work has been further developed by Brown [22], Lee [46], and Iversen [38]. Common to all these systems is that the goal is to infer a type for a self-contained action for use in code generation. This differs from what we will present in this chapter in that we want to type check semantic functions where the embodied action describes a small part of a full program, and the main goal is to give the language developer useful feedback about his description and let him test assertions about semantic functions.

Doh and Schmidt [30] describe a method for extracting typing laws from semantic functions. This is not type checking of the semantic functions, but a way to compute type rules for the described language from the semantic functions.

In [95] Ørbæk describes a soft type inference algorithm for semantic functions. The algorithm is not dependent on the user giving any kind of type annotations, like signatures, to the semantic functions; instead it infers a type by looking at all the semantic equations in the language description. This differs from our approach because we want to type check the semantic equations that describe a single language construct without looking at the whole language description.

7.3 Type system

Our type system consists of a set of types $\mathcal{T}$ ordered by a subtype relation and a set of type rules that can be used to derive a proof that an action has a certain

type. We will present both in the following two sub-sections. Throughout the rest of this chapter we will use τ as a variable that ranges over types.

7.3.1 Types

We shall view types as sets of values. Our type system has three different kinds of types: the built-in AN types, the action types, and the user defined types. The built-in AN types are listed in Fig. 7.1.

$$\begin{array}{rcl} \mathit{Type} & ::= & \mathsf{data} \mid \mathsf{datum} \mid \emptyset \mid \mathsf{integer} \mid \mathsf{boolean} \mid \mathsf{token} \mid \mathsf{bindable} \\ & & \mathsf{bindings} \mid \mathsf{storable} \mid \mathsf{cell} \mid \mathit{ActionType} \mid \mathit{Type} \times \ldots \times \mathit{Type} \end{array}$$

Figure 7.1: AN built-in types

The type data contains all values, and all types are subtypes of data. All values except tuples of data is included in the type datum. The type $\emptyset$ does not contain any values. Notice that action types (*ActionType*) are also included in the built-in types; this is necessary because actions can be used as data in AN. The product type is the type of tuples of data, and the symbol () denotes the product type of length 0, the type of the empty tuple (like `unit` in Standard ML).

$$\begin{array}{rcl} \mathit{ActionType} & ::= & \mathsf{Action} \mid \mathsf{using}\ \mathit{Type} \mid \mathsf{giving}\ \mathit{Type} \mid \mathsf{raising}\ \mathit{Type} \mid \\ & & \mathsf{infallible} \mid \mathsf{closed} \mid \mathsf{terminates} \mid \mathsf{uncreative} \mid \\ & & \mathsf{ineffective} \mid \mathsf{stable} \mid \mathit{ActionType}\ \&\ \mathit{ActionType} \end{array}$$

Figure 7.2: Action types

Action types are listed in Fig. 7.2. We use the symbol & to denote the intersection of two action types. The type Action is the supertype of all action types and says nothing about the action, except that it is an action. The three types parameterised with a type, 'using τ', 'giving τ', and 'raising τ', are the types for actions that can be given data of some type, actions that produce data of some type when they terminate normally, or actions that produce data of some type in case of abrupt termination, respectively.

An action type which does not contain 'using τ', 'giving τ', or 'raising τ' is equal to the same action type with 'using $\emptyset$', 'giving data', or 'raising data' respectively added (this means that 'Action $\equiv$ using $\emptyset$ & giving data & raising data'). This is also illustrated in the equivalence in Fig. 7.3. This equivalence is a consequence of 'using τ' being contravariant in its type argument and 'giving τ' and 'raising τ' being covariant, as shown in the subtype relations listed in Fig. 7.4. Throughout the rest of this chapter we shall use α as a variable to range over all action types and γ to range over atomic action types (all action types listed in Fig. 7.2 except '*ActionType* & *ActionType*').

The type 'using data' contains only the actions which accept all types of input. Many of the actions with this type ignore their input, like 'result D'.

The types 'giving $\emptyset$' and 'raising $\emptyset$' contain the actions that cannot terminate normally or abruptly, respectively.

The names of the rest of the types should indicate what their intended meaning is. To illustrate their use, the action type 'giving token $\times$ bindings & infallible & stable' describes the actions which produce a pair consisting of a token and bindings, and do not fail or inspect memory. The action type 'using storable & closed & ineffective & uncreative' describes actions which can be given a storable, are closed with respect to bindings, do not update storage, and do not allocate new memory locations.

Some of the action types are "negative" in the way that they describe behaviour an action may *not* have: it may *not* fail (infallible), it may *not* create new memory cells (uncreative), it may *not* update memory (ineffective), or it may *not* inspect memory cells (stable). The reason we have chosen "negative" types in these cases is that it is difficult (often impossible) to determine if an action fails or manipulates storage. It is difficult because we cannot with static analysis determine which parts of an action are evaluated. On the other hand we can easily point out a large set of actions that, for instance, do not create memory cells (the actions that do not contain the action create). This also means that if an action type does not contain, for instance, infallible, it describes all actions that might fail or not fail.

Fig. 7.3 presents an equivalence on action types. The five rules say that the order of the atomic action types is not important, the action types 'using $\emptyset$', 'giving data', and 'raising data' can be introduced, and if there is a subtype relation between two atomic action types the "highest" type can be removed (this also means that if an atomic action type occurs twice in an action type one of the occurrences can be removed). When a type operator or a type rule mentions an action type, we shall assume that the equivalence has been applied to the action type such that the type operator or the type rule can be applied.

$$\gamma_1 \,\&\, \ldots \,\&\, \gamma_{i-1} \,\&\, \gamma_i \,\&\, \ldots \,\&\, \gamma_n \equiv \gamma_1 \,\&\, \ldots \,\&\, \gamma_i \,\&\, \gamma_{i-1} \,\&\, \ldots \,\&\, \gamma_n$$

$$\alpha \equiv \mathsf{using}\ \emptyset \,\&\, \alpha$$

$$\alpha \equiv \mathsf{giving\ data} \,\&\, \alpha$$

$$\alpha \equiv \mathsf{raising\ data} \,\&\, \alpha$$

$$\gamma_1 \,\&\, \gamma_2 \,\&\, \ldots \,\&\, \gamma_n \equiv \gamma_2 \,\&\, \ldots \,\&\, \gamma_n \quad \textit{when} \quad \gamma_2 \leq \gamma_1$$

Figure 7.3: Equivalence on action types

The readers familiar with the previous work on inferring types for actions [22, 32, 38, 46] might have noticed that our types cannot describe the bindings used by an action. In previous work the bindings used by an action were also inferred using record types. This allowed a stronger type inference because the type of the output from yielders, like 'bound-to the token x', was more specific than just bindable, and the type inference algorithm was able to check that the

token was actually bound in the current bindings. Due to the fact that token values are seldom known in a semantic function before the function is applied, the type system cannot deal with bindings on a more detailed level than the atomic type bindings. To illustrate this, a semantic function containing the action 'give the bindable bound-to I' (where I is an ASDF variable ranging over tokens) will always type check in our system because the value of I is not known until the semantic function is applied, so we cannot check that the instantiation of I is bound in the current bindings.

In an ASDF module the user can provide type information. A production, like '*Bindable* ::= *Integer*' defines the type integer to be a subtype of bindable[1]. A more advanced production, like '*Func* ::= func(action : using val & giving val)' (see Module 18 on page 57), defines the data constructor func to be a data operator which takes an action of type 'using val & giving val & raising val' and gives data of type func. The production also defines the data selector action to be a data operator which takes a func and gives an action of the before mentioned type. Finally ASDF modules can also contain signatures of semantic functions, like 'evaluate : *Exp* → Action & using data & giving val' (see Module 2 on page 50).

As mentioned before, the set of types $\mathcal{T}$ is ordered, and the ordering $\leq$ is defined in Fig. 7.4.

7.3.2 Type rules

Type rules can be used to construct a proof that an action has a certain type, and from type rules type inference rules can be constructed. For an algorithm that checks that an action has a certain type, see Section 7.6.

In Fig. 7.6 we see examples of type rules for the actions used to describe normal flow of data and control in programming languages. The rules are conditional as illustrated in rule 7.2 where the premises state the types of the two subactions A_1 and A_2. In all type rules for action combinators the premises will state what the types of the subactions are. Rule 7.2 also has other conditions which state that the type of the data produced by A_1 should not be $\emptyset$ (recall that 'giving $\emptyset$' means that the action does not terminate normally and then the right subaction would never be executed, which we consider an error). The condition '$\tau_1' \leq \tau_2$' states that the type of data produced by A_1 should be a subtype of the type of data that can be given to A_2. If the premises hold, we can derive the type of 'A_1 then A_2' using the types from the premises and appropriate type operators to combine them. The $\cup$ ($\cap$) operator computes the union (intersection) of two types, and the $\cup_{ac}$ operator takes two action types and returns the intersection of the atomic action types occurring in both the action types. In other words '$\alpha_1 \cup_{ac} \alpha_2$' is the lowest action type bigger than α_1 and α_2, i.e., a union on action types rounded up to the nearest action type (a least upper bound).

The rule for the action combinator and (rule 7.3) introduces the type operator $\oplus$ which concatenates two types into a product type. A formal definition

[1]The convention in ASDF is to use words starting with capital letters for naming syntactic sorts whereas AN uses small letters in types

$\tau \leq \tau$

$\tau_1 \leq \tau_2 \ \wedge \ \tau_2 \leq \tau_3 \ \Rightarrow \ \tau_1 \leq \tau_3$

$\emptyset \leq \tau$

$\tau \leq \mathsf{datum} \quad \textit{when} \quad \tau \neq () \wedge \forall n \geq 2, \tau_i.\ \tau \neq \tau_1 \times \ldots \times \tau_n$

$\tau \leq \mathsf{data}$

$\alpha \leq \mathsf{Action}$

$\alpha \leq \gamma_1 \ \& \ \ldots \ \& \ \gamma_n \quad \textit{when} \quad \forall i \in 1..n.\ \alpha \leq \gamma_i$

$\gamma_1 \ \& \ \ldots \ \& \ \gamma_n \leq \gamma \quad \textit{when} \quad \exists i \in 1..n.\ \gamma_i \leq \gamma$

$\mathsf{using}\ \tau_1 \leq \mathsf{using}\ \tau_2 \quad \textit{when} \quad \tau_2 \leq \tau_1$

$\mathsf{giving}\ \tau_1 \leq \mathsf{giving}\ \tau_2 \quad \textit{when} \quad \tau_1 \leq \tau_2$

$\mathsf{raising}\ \tau_1 \leq \mathsf{raising}\ \tau_2 \quad \textit{when} \quad \tau_1 \leq \tau_2$

$\tau_1 \times \ldots \times \tau_n \leq \tau_1' \times \ldots \times \tau_n' \quad \textit{when} \quad \forall i \in 1..n.\ \tau_i \leq \tau_i'$

+ user defined relations in ASDF modules

Figure 7.4: Subtype relation

$$\mathsf{simple} = \mathsf{infallible}\ \&\ \mathsf{closed}\ \&\ \mathsf{terminates}\ \&\ \mathsf{uncreative}\ \&\ \mathsf{ineffective}\ \&\ \mathsf{stable} \tag{7.1}$$

Figure 7.5: Definition of simple

of all the type operators can be found in Fig. 7.7.

The rules for give (rule 7.7 and 7.8) use the constant simple which is an abbreviation for an action type. The expansion can be found in rule 7.1 in Fig. 7.5. The type of give O depends on the signature of the data operator O where the question mark indicates that it is a partial operator. The action 'check O' (rule 7.9) also contains a data operator, but the rule does not depend on whether the operator is partial since 'check O' can still terminate abruptly when the data operator is not partial. The rule insists that the result type of the operator is boolean.

Each rule has an action type on the left hand side of the turnstile, and

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \tau_1' \leq \tau_2,\ \tau_1' \neq \emptyset\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{then}\ A_2 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_2'\ \& \\ \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{7.2}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{and}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \oplus \tau_2')\ \& \\ \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{7.3}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{and\text{-}then}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \oplus \tau_2')\ \& \\ \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{7.4}$$

$$\alpha_u \vdash \mathsf{copy} : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \tau\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \tag{7.5}$$

$$\frac{D : \tau}{\alpha_u \vdash \mathsf{result}\ D : \mathsf{using\ data}\ \&\ \mathsf{giving}\ \tau\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple}} \tag{7.6}$$

$$\frac{O : \tau \rightarrow ?\ \tau'}{\alpha_u \vdash \mathsf{give}\ O : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \tau'\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple}} \tag{7.7}$$

$$\frac{O : \tau \rightarrow \tau'}{\alpha_u \vdash \mathsf{give}\ O : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \tau'\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple}} \tag{7.8}$$

$$\frac{O : \tau \rightarrow \mathsf{boolean}}{\alpha_u \vdash \mathsf{check}\ O : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \tau\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple}} \tag{7.9}$$

$$\frac{\alpha_u \vdash A : \alpha}{\alpha_u \vdash \mathsf{indivisibly}\ A : \alpha} \tag{7.10}$$

$$\alpha_u \vdash \mathsf{choose\text{-}nat} : \mathsf{using\ data}\ \&\ \mathsf{giving\ integer}\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \tag{7.11}$$

$$\frac{\alpha' \vdash A : \alpha'}{\alpha_u \vdash \mathsf{unfolding}\ A : \alpha'} \tag{7.12}$$

$$\frac{\mathsf{terminates} \notin \alpha_u}{\alpha_u \vdash \mathsf{unfold} : \alpha_u} \tag{7.13}$$

Figure 7.6: Type rules for normal flow of data and control AN

most of the rules just propagate it to the premises. The action type is used in connection with the two actions related to unfolding (rule 7.12 and 7.13). The

Type concatenation

$$\begin{array}{rcl} \tau_1 \oplus \tau_2 & = & \tau_1 \times \tau_2 \\ \emptyset \oplus \tau & = & \emptyset \\ \tau \oplus \emptyset & = & \emptyset \end{array}$$

Action type combinator

$$\begin{array}{rcll} (\gamma \ \& \ \alpha_1) \cup_{ac} (\gamma \ \& \ \alpha_2) & = & \gamma \ \& \ (\alpha_1 \cup_{ac} \alpha_2) & \\ (\gamma \ \& \ \alpha_1) \cup_{ac} \alpha_2 & = & (\alpha_1 \cup_{ac} \alpha_2) & \textit{when} \quad \gamma \notin \alpha_2 \\ \gamma \cup_{ac} (\gamma \ \& \ \alpha) & = & \gamma & \\ \gamma \cup_{ac} \alpha & = & \mathsf{Action} & \textit{when} \quad \gamma \notin \alpha \end{array}$$

Action type subtraction

$$\begin{array}{rcll} (\gamma \ \& \ \alpha) \backslash \gamma & = & \alpha \backslash \gamma & \\ (\gamma_1 \ \& \ \alpha) \backslash \gamma_2 & = & \gamma_1 \ \& \ (\alpha \backslash \gamma_2) & \textit{when} \quad \gamma_1 \neq \gamma_2 \\ \gamma \backslash \gamma & = & \mathsf{Action} & \\ \gamma_1 \backslash \gamma_2 & = & \gamma_1 & \textit{when} \quad \gamma_1 \neq \gamma_2 \end{array}$$

Action type projections

$$has(\gamma, \alpha) = \begin{cases} \gamma & \textit{when} \quad \gamma \in \alpha \\ \mathsf{Action} & \textit{when} \quad \gamma \notin \alpha \end{cases}$$

Figure 7.7: Type operators

type of 'unfolding A' is the same as the type of A, and the type of unfold is the same as the type of the enclosing unfolding action. When using the rule for unfolding, the type of A must be guessed and then passed on to the premise that derives the type for A. The rule for unfold just states that unfolds's type is the type left to the turnstile and that this type cannot contain terminates.

The rules in Fig. 7.8 concern the actions used to describe exceptional and alternative control flow (like raising exceptions and conditional expressions). Comparing with Fig. 7.6 we see that there are many similarities (compare throw with copy, then with catch, and and with and-catch). The main difference is that some actions terminate abruptly instead of normally.

Intersection between action types is very common in our type system, but as illustrated in the rule for fail (rule 7.17), there is also a subtraction operator \. The action fail does of course fail, and therefore we must remove the type infallible from its type.

In rule 7.18 we introduce the type operator *has*. The domain of *has* is an atomic action type and an action type. The operator returns the first type if the second action type contains the first, otherwise the result is Action. Using this operator ensures that the type of the whole action contains infallible if the right subaction cannot fail.

In Fig. 7.9 the type rules for actions describing declarations are shown. The atomic type bindings is used in all three rules to describe that an action produces a mapping from token's to data. As discussed in Subsection 7.3.1, action types does not describe the bindings used by an action, but the type closed indicates that an action does not use the current bindings.

$$\alpha_u \vdash \mathsf{throw} : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \emptyset\ \&\ \mathsf{raising}\ \tau\ \&\ \mathsf{simple} \tag{7.14}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \tau_1^r \leq \tau_2,\ \tau_1^r \neq \emptyset\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{catch}\ A_2 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \mathsf{raising}\ \tau_2^r\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{7.15}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{and\text{-}catch}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \mathsf{raising}\ (\tau_1^r \oplus \tau_1^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{7.16}$$

$$\alpha_u \vdash \mathsf{fail} : \mathsf{using}\ \mathsf{data}\ \&\ \mathsf{giving}\ \emptyset\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{infallible} \tag{7.17}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \mathsf{infallible} \notin \alpha_1\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{else}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\ \&\ \mathit{has}(\mathsf{infallible}, \alpha_2)\end{array}} \tag{7.18}$$

Figure 7.8: Type rules for abrupt and alternative control flow AN

$$\frac{\alpha_u \vdash A : \mathsf{giving}\ \mathsf{bindings}\ \&\ \alpha'}{\alpha_u \vdash \mathsf{recursively}\ A : \mathsf{giving}\ \mathsf{bindings}\ \&\ \alpha'} \tag{7.19}$$

$$\alpha_u \vdash \mathsf{copy\text{-}bindings} : \mathsf{using}\ \mathsf{data}\ \&\ \mathsf{giving}\ \mathsf{bindings}\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{closed} \tag{7.20}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \mathsf{bindings}\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{r}\alpha_u \vdash A_1\ \mathsf{scope}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ \tau_2'\ \& \\ \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\ \&\ \mathit{has}(\mathsf{closed}, \alpha_1)\end{array}} \tag{7.21}$$

Figure 7.9: Type rules for declarative AN

Actions can handle other actions as data; this necessitates the inclusion of action types in the set of ordinary types so it becomes a higher-order type system. In Fig. 7.10 this is illustrated. The actions there expect an action as input and either execute it with some arguments and return the result (rule 7.22),

$$\frac{\tau_2 \leq \tau_1}{\alpha_u \vdash \mathsf{apply} : \mathsf{using}\ ((\mathsf{using}\ \tau_1\ \&\ \alpha') \times \tau_2)\ \&\ \alpha' \setminus \mathsf{terminates}} \quad (7.22)$$

$$\alpha_u \vdash \mathsf{close} : \mathsf{using}\ \alpha'\ \&\ \mathsf{giving}\ (\alpha'\ \&\ \mathsf{closed})\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \quad (7.23)$$

Figure 7.10: Type rules for reflective AN

or they return a moderated action (rule 7.23). We cannot guarantee that the action apply terminates because it might recur forever, and therefore terminates must be removed from α'. Notice also that the use of the variable α' expresses how the type of apply depends on the type of the action given to apply.

$$\alpha_u \vdash \mathsf{create} : \mathsf{using}\ \mathsf{storable}\ \&\ \mathsf{giving}\ \mathsf{cell}\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{uncreative} \quad (7.24)$$

$$\alpha_u \vdash \mathsf{update} : \mathsf{using}\ (\mathsf{cell} \times \mathsf{storable})\ \&\ \mathsf{giving}\ ()\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple} \setminus \mathsf{ineffective} \quad (7.25)$$

$$\alpha_u \vdash \mathsf{inspect} : \mathsf{using}\ \mathsf{cell}\ \&\ \mathsf{giving}\ \mathsf{storable}\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple} \setminus \mathsf{stable} \quad (7.26)$$

Figure 7.11: Type rules for imperative AN

The three rules in Fig. 7.11 shows the use of the action types uncreative, ineffective, and stable. Besides defining the type of data used, produced, and raised by the actions the rules also illustrate how the three action types are closely connected to these three actions, i.e., the type of an action contains uncreative, ineffective, or stable if, and only if, it does not contain the actions create, update, or inspect, respectively.

The actions found in semantic functions can contain applications of other semantic functions as subactions (as illustrated in Module 7 on page 53). The type rule for these applications is shown in rule 7.27. The rule states that if the function f has a signature $\sigma \to \alpha$, then the result of applying f to a construct S of syntactic sort σ has type α.

The subsumption rule (rule 7.28 in Fig. 7.13) says that if an action A has a type α and α' is a supertype of α, then A also has the type α'.

AN contains only few built-in data operators and expects the user to provide the necessary definitions of data and data operators. We shall not spend many lines on data notation here, but it is relevant to know about the built-in partial data operator 'the τ' which performs type projections. Given data of type τ it

$$\frac{\alpha_u \vdash f : \Sigma \to \alpha \quad S : \Sigma}{\alpha_u \vdash f\ S : \alpha} \tag{7.27}$$

Figure 7.12: Type rule for semantic function

$$\frac{\alpha_u \vdash A :\ \alpha \quad \alpha \leq \alpha'}{\alpha_u \vdash A :\ \alpha'} \tag{7.28}$$

Figure 7.13: Subsumption rule

returns the given data, otherwise it is undefined. In our type system we have to settle with the type of data given to '**the** τ' not being disjoint with τ because we often cannot determine types that are specific enough. An example of this is the action '**inspect then give the integer**' where **inspect** produces a storable which is not necessarily an **integer** (but an integer can be a **storable**). This liberal typing of '**the** τ' turns our type checker into a soft type checker, because we cannot always guarantee that the action will not err when it tries to perform '**the** τ'.

7.4 An example

To illustrate the use of the type system we will try to type check the ASDF module *Exp/Local* (Module 7 on page 53). The module describes declarations local to an expression. To maintain simplicity we have omitted '**raising** τ' and the action before the turnstile from the rules in this example. Before type checking can start, the type information defined by the user must be collected. The type information relevant for the module *Exp/Local* can be found in the modules *Exp* (Module 2 on page 50) and *Dec* (Module 26 on page 180).

It is also necessary to rewrite the action in the semantic equation to the corresponding kernel action. This is necessary because we only have type rules for kernel actions. The action

(furthermore (declare *D*)) scope (evaluate *E*)

corresponds to the kernel action

```
(1)        ((copy-bindings and (declare D))
(2)      then
(3)        (give overriding))
(4)   scope
(5)      (evaluate E)
```

Starting from the top of the parse tree representing the kernel action we apply rule 7.21, the type rule for the action combinator **scope**. The rule requires a

type for the two subactions, so we use rule 7.2 to derive a type for the left subaction (lines 1-3), and again we must infer a type for the two subactions (lines 1 and 3). To infer a type for the action in line 1 we use rule 7.20:

$$\vdash \mathsf{copy\text{-}bindings} : \ \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}\ \&\ \mathsf{simple} \setminus \mathsf{closed} \tag{7.29}$$

and rule 7.27

$$\frac{\begin{array}{l}\vdash \mathsf{declare} : \mathit{Dec} \to \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings} \\ \mathit{D} : \mathit{Dec}\end{array}}{\vdash \mathsf{declare}\ \mathit{D} : \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}} \tag{7.30}$$

(where we use the signature from Module 26 on page 180 to satisfy the premises) and finally rule 7.3 (7.29 together with 7.30 provide a proof for the premises).

$$\frac{\begin{array}{l}\vdash \mathsf{copy\text{-}bindings} : \ \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}\ \&\ \mathsf{simple} \setminus \mathsf{closed} \\ \vdash \mathsf{declare}\ \mathit{D} : \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}\end{array}}{\begin{array}{l}\vdash \mathsf{copy\text{-}bindings\ and\ (declare}\ \mathit{D}\mathsf{)} : \\ \qquad \mathsf{using\ data}\ \&\ \mathsf{giving\ (bindings, bindings)}\end{array}} \tag{7.31}$$

The data operator overriding has signature

$$\mathsf{overriding} : \mathsf{bindings} \times \mathsf{bindings} \to \mathsf{bindings} \tag{7.32}$$

and using rule 7.8 we get

$$\frac{\mathsf{overriding} : \mathsf{bindings} \times \mathsf{bindings} \to \mathsf{bindings}}{\begin{array}{l}\vdash \mathsf{give\ overriding} : \\ \qquad \mathsf{using\ (bindings} \times \mathsf{bindings)}\ \&\ \mathsf{giving\ bindings}\ \&\ \mathsf{simple}\end{array}} \tag{7.33}$$

Now combining 7.31, 7.33, and rule 7.2 we get:

$$\frac{\begin{array}{l}\vdash A_1 : \mathsf{using\ data}\ \&\ \mathsf{giving\ (bindings, bindings)} \\ \vdash \mathsf{give\ overriding} : \mathsf{using\ (bindings,\ bindings)}\ \& \\ \qquad\qquad\qquad\qquad \mathsf{giving\ bindings}\ \&\ \mathsf{simple} \\ \mathsf{bindings} \times \mathsf{bindings} \leq \mathsf{bindings} \times \mathsf{bindings}, \\ \mathsf{bindings} \times \mathsf{bindings} \neq \emptyset\end{array}}{\vdash A_1\ \mathsf{then\ (give\ overriding)} : \mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}} \tag{7.34}$$

(where A_1 is 'copy-bindings and (declare D)'). The application of the semantic function evaluate in line 5 can be typed using rule 7.27:

$$\frac{\begin{array}{l}\vdash \mathsf{evaluate} : \mathit{Exp} \to \mathsf{using\ data}\ \&\ \mathsf{giving\ val} \\ \mathit{E} : \mathit{Exp}\end{array}}{\vdash \mathsf{evaluate}\ \mathit{E} : \mathsf{using\ data}\ \&\ \mathsf{giving\ val}} \tag{7.35}$$

Finally we can use 7.34, 7.35, and rule 7.21 to derive a type for the whole action (lines 1-5).

$$\frac{\vdash A_{1-3} : \mathsf{using\ data\ \&\ giving\ bindings} \quad \vdash \mathsf{evaluate}\ E : \mathsf{using\ data\ \&\ giving\ val}}{\vdash A_{1-3}\ \mathsf{scope\ (evaluate}\ E\mathsf{)} : \mathsf{using\ data\ \&\ giving\ val}} \tag{7.36}$$

where A_{1-3} is the part of the whole action that spans lines 1-3. We now have a type for the action from the right-hand side of the semantic equation in the module *Exp/Local*, and we can conclude the type check by checking that the inferred type is a subtype of the action type in the signature for evaluate. Since they are both 'using data & giving val', we conclude that the semantic equation type checks.

7.5 Constructive type checking

A constructive ASD of a programming language written in ASDF is extensible and reusable. This is advantageous because it allows incremental development of descriptions, e.g., we can start by describing the core of a language and then incrementally add more features to the language by adding more modules. Furthermore the modules can be reused by reference in other language descriptions.

This section deals with the problem that we might want different signatures for the same semantic function depending on which properties we want, for instance, expressions to have in our description. The problem is complicated further because we want the ASDF modules to preserve their reusability.

In a typical description of a language we have a module *Exp* containing all the features common to expressions as illustrated in Module 2 on page 50. This module is then imported (automatically) from all modules describing expressions. In *Exp* we put the signature of the semantic function evaluate which maps expressions to actions. The signature requires that the action resulting from applying evaluate to an expression can be given any data and normally produces a value. If we for instance are describing a purely functional language, we might want to check whether the modules we include have side-effects. Therefore we would need signatures that include the types uncreative, ineffective, and stable. Modules, like *Exp*, should be fixed so that they can be reused in language descriptions, so changing the signature in *Exp* is not an option. Two solutions to the problem can be envisaged:

1. Before every type check, the user gives a signature of the function which is the target of the type checking, and the type checker infers the signatures of the other semantic functions employed in the semantic function.

2. Before every type check, the user specifies a module that contains extra type info for use in connection with type checking a particular module.

The advantage of the first suggestion is that it allows the user to see which demands it makes on the employed semantic functions when he makes demands on a semantic function. The disadvantage is that it is more difficult to implement because we have to do type inference instead of just type checking.

The second suggestion is easier to implement. The extra module given to the type checker contains more specialised versions of signatures, for instance, one could have a module that can be used to check that a module is purely functional. We have chosen this solution for our implementation.

When having more than one signature for the same semantic function that only differ with respect to the output, our type checker merges the signatures as illustrated here:

$$\begin{array}{l}\mathsf{evaluate}\colon \mathit{Exp} \to \alpha_1\\ \mathsf{evaluate}\colon \mathit{Exp} \to \alpha_2\end{array}$$

is merged into

$$\mathsf{evaluate}\colon \mathit{Exp} \to \alpha_1 \;\&\; \alpha_2$$

and the equivalence (Fig. 7.3) is used to simplify the action type 'α_1 & α_2'.

7.6 Implementation

Type inference rules for copy, unfolding, and apply, involve type variables. This means that implementing the type system is more complicated than a depth-first traversal of the parse tree where the type rules are used to construct a type. The problem with rule 7.5, 7.12, and 7.22 is that they involve guessing types. Our implementation uses type inclusion constraints on type schemes (types with type variables). The types τ are extended with type variables θ.

To keep the implementation simple we shall use another representation of action types. The action type presented in the previous sections is readable and useful in semantic function signatures, but the following is better in an implementation, because it does not need to be normalised and type inclusion constraints with action types can more easily be simplified (see Fig. 7.14). We shall use the type

$$at(\tau, \tau, \tau, \tau', \tau', \tau', \tau', \tau', \tau')$$

The action type constructor at has nine arguments, one for each atomic action type, and is equivalent to the action type presented in the previous sections. The first three arguments can contain arbitrary types, and represent 'using τ', 'giving τ', and 'raising τ'. The last six arguments can contain empty, data, or a type variable θ. The arguments represent infallible, closed, terminates, uncreative, ineffective, and stable in that order. This means that an action type like 'using integer & giving boolean & closed & terminates & ineffective' is represented by the type

$$at(\mathsf{integer}, \mathsf{boolean}, \emptyset, \mathsf{data}, \emptyset, \emptyset, \mathsf{data}, \emptyset, \mathsf{data})$$

We use $\emptyset$ to indicate that the atomic action type corresponding to an argument is present and data means that it is absent. The type variables are used if the algorithm cannot determine whether an atomic action type is present or

absent. The action type operator $\cup_{ac}$ produces an action type where each argument is the union of the types in the same argument in the two given action types. The type operator $\setminus$ sets the appropriate argument in an action type to data.

The idea behind the algorithm is that it transforms a set of constraints to a set of constraints in *inductive* form or an inconsistent set of constraints. Inductive sets of constraints have solutions [2]. A constraint is inductive if it has the form $\theta_j \subseteq \tau$ or $\tau \subseteq \theta_j$, and the set of variables on the top level in τ is included in $\{\theta_1, \ldots, \theta_{j-1}\}$ (see Theorem 7.2 in [2]). Here we have assumed that the type variables are numbered. The algorithm is:

1. Collect type information from the ASDF modules.
2. Traverse the parse tree of the action while generating constraints.
3. Repeat the following steps until all constraints in S are inductive and no additional inductive constraints can be added:
   - For any constraint that is not inductive, apply the lowest numbered applicable rule in Fig. 7.14 to simplify the set of constraints S.
   - For any pair of inductive constraints '$\tau_1 \subseteq \theta$' and '$\theta \subseteq \tau_2$' in S, add the constraint '$\tau_1 \subseteq \tau_2$' to S.
   - Stop if S is no longer consistent.
4. If we are not able to apply a type rule for each node in the parse tree, or the final set of constraints is not consistent, the action does not type check.

The constraints generated in point 2 are type inclusion constraints ($\subseteq$) and come from the use of $\leq$ in the type rules. Occurrences of the subtype relation '$\tau_1 \leq \tau_2$' (see Rules 7.2 and 7.15) are translated into '$\tau_1 \subseteq \tau_2$'.

Notice that '$\tau_1 \subseteq \tau_2$' is a constraint used in the implementation, and the interpretation is that the constraint holds if '$\tau_1 \leq \tau_2$' with the right assignment of types to the variables in τ_1 and τ_2. In the algorithm we are not going to find an assignment of types to all variables such that all constraints hold; instead we check whether an assignment exists.

The rules in Fig. 7.14 can be applied to the set of constraints to simplify the constraints. It is essential that the constraints on the left hand side of $\equiv$ hold for a given substitution of types to type variables if, and only if, the constraints on the right hand side do. The first rule removes the obvious constraint that does not add any extra information. Rule 2 simplifies a constraint with to product types of equal length by generating constraints comparing all of the element types. The two next rules use well known results from set theory to remove union and intersection of types that cannot be normalised. Notice that we do not have equivalent rules for intersection or union on the other side of the $\subseteq$. Such rules will not be used, as the reader can convince himself about by looking at the type rules. Union (intersection) of types only occurs as the type of output from (input to) actions, and the constraints are always generated

(1) $S \cup \{\emptyset \subseteq \tau\} \equiv S$

(2) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1\} \cup \ldots \cup \{\tau_n \subseteq \tau'_n\}$

(3) $S \cup \{\tau_1 \subseteq \tau_2 \cap \tau_3\} \equiv S \cup \{\tau_1 \subseteq \tau_2, \tau_1 \subseteq \tau_3\}$

(4) $S \cup \{\tau_1 \cup \tau_2 \subseteq \tau_3\} \equiv S \cup \{\tau_1 \subseteq \tau_3, \tau_2 \subseteq \tau_3\}$

(5) $S \cup \{\tau_1 \times \ldots \times \tau_m \oplus \theta \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1, \ldots, \tau_m \subseteq \tau'_m, \theta \subseteq \tau'_{m+1} \times \ldots \times \tau'_n\}$ *when* $m \leq n$

(6) $S \cup \{\theta \oplus \tau_1 \times \ldots \times \tau_m \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\theta \subseteq \tau'_1 \times \ldots \times \tau'_{n-m}, \tau_1 \subseteq \tau'_{n-m+1}, \ldots, \tau_m \subseteq \tau'_n\}$ *when* $m \leq n$

(7) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \tau'_1 \times \ldots \times \tau'_m \oplus \theta\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1, \ldots, \tau_m \subseteq \tau'_m, \tau_{m+1} \times \ldots \times \tau_n \subseteq \theta\}$ *when* $m \leq n$

(8) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \theta \oplus \tau'_1 \times \ldots \times \tau'_m\} \equiv$
$S \cup \{\tau_1 \times \ldots \times \tau_{n-m} \subseteq \theta, \tau_{n-m+1} \subseteq \tau'_1, \ldots, \tau_n \subseteq \tau'_m\}$ *when* $m \leq n$

(9) $S \cup \{at(\tau_1, \tau_2, \ldots, \tau_9) \subseteq at(\tau'_1, \tau'_2, \ldots, \tau'_9) \equiv$
$S \cup \{\tau'_1 \subseteq \tau_1, \tau_2 \subseteq \tau'_2, \ldots, \tau_9 \subseteq \tau'_9\}$

Figure 7.14: Constraint simplification

by requiring that the output of one action be a subtype of the input given to another action (in rules 7.2 and 7.15). Rules 5 to 8 simplify constraints where application of the concatenation operator could not be normalised. In the last rule action types are removed from the set of constraints. Notice the covariance in the first argument of the action type which reflects the covariance in the atomic action type '$\mathsf{using}\ \tau$' expressed in the subtype relations.

A set of constraints is inconsistent if it contains '$\tau_1 \subseteq \tau_2$' and '$\tau_1 \not\leq \tau_2$'. If the constraints contains '$\tau_1 \oplus \tau_2 \subseteq \tau_3$' or '$\tau_1 \subseteq \tau_2 \oplus \tau_3$' where at least two of the τ's are type variables, the simplification rules cannot simplify the constraint to a set of inductive constraints. To be on the safe side we will also consider constraint sets containing these cases to be inconsistent.

Our algorithm is almost identical to the one found in [2], so we shall not bother proving that the inductive set of constraints resulting from the simplifications has a solution if, and only if, the original set of constraints has a solution. The algorithm does not calculate a type for an action, but instead it checks that a type exist. This is sufficient because we just want to know whether an action in a semantic equation has a type and that this type is a subtype of the action type found in the signature of the corresponding semantic

function.

In the Action Environment the type checker is invoked over a module, and the environment then collects type information from all imported modules before passing the semantic equations in the module together with the type information to the type checker. The result is either a message indicating that type check went well or error messages specifying where problems have been identified. The error messages can indicate where the type checker failed to apply a type rule or which action caused the constraint that made the set of constraints inconsistent. Another problem might be dead code which can occur if the left hand side of an action combinator cannot terminate in a way that allows the right hand side to be executed (for instance, if A_1 in 'A_1 catch A_2' never terminate abruptly, i.e., the type of A_1 contains 'raising $\emptyset$').

7.7 What can we prove?

It would be interesting to prove that the type checker can say something interesting about an action. For a normal type checker, we would want to prove that an action that type checks is well behaved. This is not possible because our type checker is liberal enough to approve actions that are not well behaved. By well behaved we mean that the action does not err because of a type error. Instead we might consider proving that if an action does not type check, then it is not well behaved, but again we run into problems. Those problems are related to the constraints that we could not simplify, and therefore resulted in an inconsistent set of constraints. The type checker rejects actions that are type correct. It appears that it is difficult to prove anything interesting about the type checker, although practical experience has shown that it is still useful.

7.8 Conclusion and future work

We have presented a type system for AS, which allows a soft type check of action semantic functions. The system has been implemented as a type checker operating on ASDF modules and as such it provides a useful tool when describing languages. The type checker supports the extensibility and reusability inherent in ASDF by letting the user supply the relevant type information before a type check.

Type checkers can almost always be improved to accept a bigger set of legal programs; this also holds for our semantic function type checker. With respect to the user-friendliness of the type checker, it is worth considering whether our type system can become more transparent [24, page 7]; can the user predict whether a semantic function will type check, and will he understand why it does not.

Chapter 8

Interpreting actions

The actions of men are the best interpreters of their thoughts.
— John Locke

When describing a language, it is convenient to be able to interpret test programs written in the language. If a description defines a particular language construct, running a test program that utilises the construct can reveal possible flaws in the definition.

There already exist interpreters of AN. The Actress system included an interpreter (Section 4.5 in [71]) that implemented a big step operational semantics of the old version of AN. One of the disadvantages in using big step semantics is that the description of interleaving becomes more complicated. In Actress they solved this by limiting the subset of AN handled by the interpreter. The action 'A_1 and A_2' (A_1 is evaluated interleaved with A_2) has the same behaviour as 'A_1 and-then A_2' (where A_1 is evaluated before A_2), and the actions describing interactive processes are not handled at all.

The interpreter in Abaco [49] is not well documented, but from testing it we conclude that it is based on the old version of AN, and it has the same limitations with respect to interleaving as Actress does.

Tijs van der Storm [83] used the ASF+SDF Meta-Environment to implement a set of action tools and among them also an interpreter based on the MSOS of AN-2 [62]. Except for some minor syntactic details, the implementation closely imitates the MSOS rules. This means that he avoids the problems with interleaving present in previous work, but it also results in a relatively slow interpreter due to the small step style. We have not been able to test the interpreter to justify the claim of slowness, but in [83] van der Storm documents a test where calculating the 20th Fibonacci number takes 300 seconds when using his evaluator. Our interpreter uses 2.5 seconds for the same computation using comparable hardware. The action that calculates Fibonacci numbers uses most of the action combinators from AN.

We have also chosen to use ASF for implementing our interpreter, because it makes the implementation fast to write, easy to read, and easy to interface with the Action Environment. Contrary to van der Storm's implementation we have chosen to use a big step style, and like in the Actress system we do not handle the and combinator properly. Instead our interpreter is faster. The

interface to the Action Environment and the speed-up compared to van der Storm's interpreter is the main motivations for implementing another action interpreter.

Our action interpreter is intended for use from within the Action Environment. When reducing a program term to an action over a module M in a language specification, the action interpreter can be invoked and must be given the action together with the data types, data constants, and data operators defined in ASDF modules imported from M.

8.1 Representing state

The main part of the interpreter is a function *eval* with the signature

$$eval : Action * Data * Bindings * Environment \rightarrow Result * Environment$$

Given an action A, data D, bindings BS, and an environment E, *eval* evaluates A with the data D and bindings BS as input. The environment E contains a mapping from memory cells to data, the action A in the nearest enclosing '`unfolding` A' (used when evaluating `unfold`), and information about subtype relations, user defined data constants, and user defined data operations collected from the ASDF modules in the language specification. The user defined information in the environment is fixed throughout the evaluation, but must be carried around since ASF has no global variables. The rest of the environment can change, and therefore the result of *eval* contains an updated environment together with a *Result* indicating how the action terminated (if it terminated) and with what data. A *Result* can be either '*normal Data*', '*abrupt Data*', or *failed*.

We might consider splitting the environment up in the fixed and the changeable part. Then *eval* should only return the changeable part of the environment. This might make the implementation of *eval* more efficient because the ASF evaluator should handle smaller terms, but since ASF uses ATerms [9] with maximal sub-term sharing the speed up is limited, and it would make the rules more complicated.

8.2 Actions

In this section we will present some of the ASF equations defining the function *eval*. The rest of the equations can be found in Appendix D.

In the rules we use variables D to range over datum and tuples of data (in this chapter just called data), D^* to range over comma separated sequences of data, BS to range over bindings, E to range over environments, R to range over results, A to range over actions, and DO to range over data operations. Notice that in ASF $< v_1, v_2 >$ denotes a pair.

The actions that describe flow of data and control are straight forward to implement (see Fig. 8.1). Applied to the action `copy` (Equation 8.1), data D, bindings BS, and environment E the interpreter results in normal termination

$$\text{[copy]} \quad \mathit{eval}(\mathsf{copy},\ D,\ BS,\ E) = <\mathit{normal}\ D,\ E> \tag{8.1}$$

$$\text{[result]} \quad \mathit{eval}(\mathsf{result}\ D1,\ D2,\ BS,\ E) = <\mathit{normal}\ D1,\ E> \tag{8.2}$$

$$\text{[give]} \quad \frac{R := \mathit{eval\text{-}dataop}(DO,\ D,\ E)}{\mathit{eval}(\mathsf{give}\ DO,\ D,\ BS,\ E) = <R,\ E>} \tag{8.3}$$

$$\text{[then]} \quad \frac{\begin{array}{l}<\mathit{normal}\ D2,\ E2> := \mathit{eval}(A1,\ D1,\ BS,\ E1),\\ <R,\ E3> := \mathit{eval}(A2,\ D2,\ BS,\ E2)\end{array}}{\mathit{eval}(A1\ \mathsf{then}\ A2,\ D1,\ BS,\ E1) = <R,\ E3>} \tag{8.4}$$

$$\text{[default-then]} \quad \frac{\begin{array}{l}<R,\ E2> := \mathit{eval}(A1,\ D1,\ BS,\ E1),\\ \mathit{normal}\ D2\ !:=\ R\end{array}}{\mathit{eval}(A1\ \mathsf{then}\ A2,\ D1,\ BS,\ E1) = <R,\ E2>} \tag{8.5}$$

$$\text{[and]} \quad \frac{\begin{array}{l}<\mathit{normal}\ D2,\ E2> := \mathit{eval}(A1,\ D1,\ BS,\ E1),\\ <\mathit{normal}\ D3,\ E3> := \mathit{eval}(A2,\ D1,\ BS,\ E2)\end{array}}{\mathit{eval}(A1\ \mathsf{and}\ A2,\ D1,\ BS,\ E1) = <\mathit{normal}\ D2{+}D3,\ E3>} \tag{8.6}$$

$$\text{[unfolding]} \quad \frac{E2 := \mathit{set\text{-}unfold\text{-}action}(E1,\ A)}{\mathit{eval}(\mathsf{unfolding}\ A,\ D,\ BS,\ E1) = \mathit{eval}(A,\ D,\ BS,\ E2)} \tag{8.7}$$

$$\text{[unfold]} \quad \frac{A := \mathit{get\text{-}unfold\text{-}action}(E)}{\mathit{eval}(\mathsf{unfold},\ D,\ BS,\ E) = \mathit{eval}(A,\ D,\ BS,\ E)} \tag{8.8}$$

$$\text{[throw]} \quad \mathit{eval}(\mathsf{throw},\ D,\ BS,\ E) = <\mathit{abrupt}\ D,\ E> \tag{8.9}$$

$$\text{[fail]} \quad \mathit{eval}(\mathsf{fail},\ D,\ BS,\ E) = <\mathit{failed},\ E> \tag{8.10}$$

Figure 8.1: Definition of *eval* for flow of data and control AN

with the data D (*normal* D) and the unchanged environment E. Equation 8.2 describing the action '**result** $D1$' is just as simple; it just returns the data $D1$

[copy-bindings] $eval(\textsf{copy-bindings}, D, BS, E) = <normal\ BS, E>$ (8.11)

[recursively] $eval(\textsf{recursively}\ A, D, BS, E) = eval(\textsf{furthermore bind}(\text{REC}, A)\ \textsf{scope}\ A, D, BS, E)$ (8.12)

[update] $$\frac{normal\ D2 := eval\text{-}dataop(\textsf{the storable}, D1, E),\ E2 := update\text{-}cell(C, D1, E1)}{eval(\textsf{update}, (C, D1), BS, E1) = <normal\ (), E2>}$$ (8.13)

[default-update] $eval(\textsf{update}, D, BS, E) = <abrupt\ (), E>$ (8.14)

[apply] $eval(\textsf{apply}, (A, D^*), BS, E) = eval(A, (D^*), \{\}, E)$ (8.15)

[close] $eval(\textsf{close}, A, BS, E) = <normal\ (\textsf{result}\ BS\ \textsf{scope}\ (\textsf{furthermore apply (the action bound-to REC, ())}\ \textsf{scope}\ A)), E>$ (8.16)

Figure 8.2: Definition of *eval* for the rest of AN

instead of the given data $D2$.

Conditional equations are used when *eval* needs to evaluate subparts of an action or update the environment. This is illustrated in Equation 8.3 where *eval* applied to '**give** DO' reduces to R, where R is the result of applying the data operator DO to the given data D (we use the function *eval-dataop* to interpret data operations). Here the result R might either indicate normal or abrupt termination.

Another example of a conditional equation is Equation 8.4 where the conditions are used to evaluate the first subaction in the action 'A_1 **then** A_2'. If it terminates normally with data $D2$, the second subaction is evaluated with $D2$ as input. The bindings are the same for the two subactions, but the environment might have changed during evaluation of $A1$. If the first subaction does not terminate normally, the alternative is defined in Equation 8.5. The condition '*normal D2* !:= R' ensures that this equation is only used when the left subaction does not terminate normally (the ASF evaluator will also try Equation 8.5 if, for instance, the right subaction cannot be evaluated in Equation 8.4).

In Equation 8.6 the conditions are also used to evaluate the two subactions, and the result is then combined in the final result using a concatenation operator ('+') on data tuples. Alternatives defining *eval*'s behaviour on the **and** combinator (see Equations D.6 and D.7) cover the cases where one of the two

subactions does not terminate normally. Notice that the evaluation order of the conditions in the ASF evaluator means that the left subaction is always evaluated before the right which means that 'A_1 and A_2' is interpreted like 'A_1 and-then A_2' (see Equation D.8), as mentioned in the beginning of this chapter.

The two actions for describing iterative control flow, unfolding and unfold, use the environment to hold the action that must be evaluated when unfold is evaluated. In Equation 8.7 the function *set-unfold-action* is used to save the body of the unfolding in the environment before evaluating the body with the updated environment. The function *set-unfold-action*, and all other auxiliary environment access functions, are also defined in ASF, but not listed here since there definitions are trivial. The saved action is looked up when evaluating the unfold action (see Equation 8.8) inside the body of an 'unfolding A' action.

The equations so far has dealt with normal flow of control. Equation 8.9 defines *eval* on the action throw, and, as expected, it is similar to the equation for copy (Equation 8.1), the only difference being the change of *normal* to *abrupt* because throw terminates abruptly and not normally as copy does. The action fail's interpretation is defined in Equation 8.10 and it is also similar to the equation for copy (here the result is *fail* instead of *normal* D). We have put the equations describing catch, and-catch, and else in the appendix because they are similar to the equations describing then and and.

A selection of the equations implementing the rest of Kernel AN is listed in Fig. 8.2.

Evaluating copy-bindings (Equation 8.11) just results in *normal BS* (where BS are the current bindings) and an unchanged environment. The equation for scope (Equation D.18) is similar to the equation for then and is therefore not displayed here.

The action for describing recursive declarations, 'recursively A' (see Equation 8.12), is evaluated by expanding it to an action that binds the special token REC to the action A so that it can be used when the closure of a recursive action is computed (see the interpretation of close in Equation 8.16). This solution is not fully satisfactory since it does not interpret actions like 'recursively (... recursively ...)' correctly. The problem is that the recursive bindings declared by the outer recursively action is not available in the body of the inner recursively. Unfortunately we have not been able to come up with a solution that interprets recursively correctly without using some static analysis of the action. The new semantics of recursively recently suggested by Mosses (see end of Section 2.4), can more easily be implemented correctly.

AN has three actions for manipulating storage and evaluating them involves changing the representation of the storage in the environment, either by allocating a new memory cell or looking up or updating the value stored in a memory cell. In Equation 8.13 the function *update-cell* is used to store datum (D) in a memory cell (C), and thereby simulating the behaviour of the update action. The result is normal termination with no data and a changed environment. The arguments for the *eval* function specifies that the action must be given a cell and datum. The function *eval-dataop* and the data operator the storable is used to check that the datum D is storable. An alternative equation is needed to handle the cases where update is not given a cell and a storable datum. Equa-

tion 8.14 defines the alternative behaviour which results in abrupt termination with no data.

For the two actions used to describe reflection, pattern matching is also used to ensure that the right type of data is given to the actions. The action apply must be given an action followed by a sequence of data as shown in Equation 8.15. Because apply evaluates the given action with the given data as input, *eval* applied to apply just rewrites to *eval* applied to the given action and the given data together with empty bindings. In the interpretation of close *eval* must construct a new action such that the given action is closed with respect to the current bindings and the recursive bindings, and this is done by combining the given action A with the action 'result BS' (that just produces the bindings BS) using the scope combinator. The action bound to the special token REC is evaluated to compute the recursive bindings. The implementation of close is not efficient because it increases the size of the given action considerably, and every time the resulting action is evaluated the action bound to REC is evaluated.

For both apply and close alternative equations similar to the one described for update exists, they are also shown in the appendix.

8.3 Types, data, and data operators

The types, data, and data operators used in an action can be both the predefined AN types, data, and data operators, as well as the user defined ones found in the ASDF modules imported from the module the action is evaluated over. In this section we shall use the variables described in the previous section together with the variables N to range over integers, TK to range over tokens, TY to range over types, $CO*$ to range over comma separated sequences of data constant and type pairs, and $STY*$ to range over comma separated sequences of pairs of types.

Data does not need to be evaluated and is just introduced using the action 'result D' as displayed in Equation 8.2.

When evaluating an action, the initial environment contains information about the user defined types and subtype relations, and the types of data constants, data constructors, and data selectors. This information is used by the function *eval-dataop* with the following signature

$$\mathit{eval\text{-}dataop} : \mathit{Action} * \mathit{Data} * \mathit{Environment} \rightarrow \mathit{Result}$$

Fig. 8.3 gives examples on how *eval-dataop* is defined.

The default behaviour for a data operation is to return the result '*abrupt* ()' indicating that the data operation could not be applied to the given data (the operation is partial). Equation 8.17 defines this behaviour.

Equations 8.18 to 8.21 concern the built-in data operators. Notice the use of ASF terms and variables to ensure that data has the right type, like the use of N in Equation 8.19 to ensure that the type projection operator is given an integer.

The user defined data operations declared in ASDF declarations like

[default-dataop] $\mathit{eval\text{-}dataop}(DO,\ D,\ E) = \mathit{abrupt}\ ()$ (8.17)

[sharp-2] $\mathit{eval\text{-}dataop}(\#\ 2,\ (D1,\ D2,\ D^*),\ E) = \mathit{normal}\ D2$ (8.18)

[the-integer] $\mathit{eval\text{-}dataop}(\mathsf{the\ integer},\ N,\ E) = \mathit{normal}\ N$ (8.19)

[more-than] $\mathit{eval\text{-}dataop}(>,\ (N1,\ N2),\ E) = \mathit{normal}\ N1 > N2$ (8.20)

[binding] $\mathit{eval\text{-}dataop}(\mathsf{binding},\ (TK,\ D),\ E) = \mathit{normal}\ \{TK : D\}$ (8.21)

[data-con]
$$\frac{\begin{array}{l}\mathit{datacon}(TY1,\ TY2) := \mathit{lookup\text{-}dataop}(DO,\ E)\\ \mathit{normal}\ D2 := \mathit{eval}(\mathsf{the}\ TY2,\ D1,\ E)\end{array}}{\mathit{eval\text{-}dataop}(DO,\ D1,\ E) = \mathit{normal}\ \mathit{data}(TY1,\ DO,\ D1)} \quad (8.22)$$

[data-sel]
$$\frac{\begin{array}{l}\mathit{datasel}(TY,\ DO2,\ DO3) := \mathit{lookup\text{-}dataop}(DO1,\ E)\\ \mathit{normal}\ D2 := \mathit{eval\text{-}dataop}(DO3,\ D1,\ E)\end{array}}{\mathit{eval\text{-}dataop}(DO1,\ \mathit{data}(TY,\ DO2,\ D1),\ E) = \mathit{normal}\ D2} \quad (8.23)$$

[the-type-1] $\mathit{eval\text{-}dataop}(\mathsf{the}\ TY, \mathit{data}(TY,\ DO,\ D),\ E) = \mathit{normal}\ \mathit{data}(TY,\ DO,\ D)$ (8.24)

[the-type-2]
$$\frac{[CO1^*,\ <D,\ TY>,\ CO2^*] := \mathit{get\text{-}constants}(E)}{\mathit{eval\text{-}dataop}(\mathsf{the}\ TY,\ D,\ E) = \mathit{normal}\ D} \quad (8.25)$$

[the-type-3]
$$\frac{\begin{array}{l}[STY1^*,\ <TY1,\ TY2>,\ STY2^*] := \mathit{get\text{-}subtypes}(E)\\ \mathit{normal}\ D2 := \mathit{eval\text{-}dataop}(\mathsf{the}\ TY2,\ D1,\ E)\end{array}}{\mathit{eval\text{-}dataop}(\mathsf{the}\ TY1,\ D1,\ E) = \mathit{normal}\ D2} \quad (8.26)$$

Figure 8.3: Definition of *eval-dataop*

Func **::=** datacons(token: *Token*, tag: *Cell*)

adds the following to the environment: '<datacons, *datacon*(func, (token, cell))>', '<token, *datasel*(func, datacons, the token #1)>', and '<tag, *datasel*(func, datacons, the cell #2)>'. In Equation 8.22 the definition of a data operator DO from

the environment is used in the evaluation of this data operator. The function *lookup-dataop* looks up the definition in the environment. If *DO* is a data constructor, *lookup-dataop* returns '*datacon*(*TY1*, *TY2*)' which means that *DO* is a data constructor that will construct data of type *TY1* when given data of type *TY2*. In the interpreter constructed data is represented as '*data*(*TY1*, *DO*, *D1*)', where *TY1* is the type of the data, *DO* is the data operator that constructed it, and *D1* is the data contained in the constructed data. But *lookup-dataop* can also return *datasel*(*TY*, *DO2*, *DO3*) which means that the data operator is a data selector that should be given constructed data of type *TY*, constructed with the data operator *DO2*, and the data operator *DO3* will then be applied to the data contained in the constructed data to select the right part of the constructed data. This is described in Equation 8.23.

Applying the type projector to constructed data is interpreted in Equation 8.24, and applying it to data constants declared in ASDF (like '`func-no-apply` : *Val*' in Module 5 on page 52) is interpreted in Equation 8.25. Notice the use of list variables (*CO**). Using list variables means that an equation in ASF can be tried several times with different instantiations of the list variables until the equation succeeds or all combinations have been tried.

The subtype relations defined in ASDF modules (like '*Val* ::= *Func*' in Module 9 on page 54) is also used by the type projection operator. Given data *D* the operator '`the` *TY1*' returns a result if *D* has type *TY1*, or if a subtype *TY2* of *TY1* exists, and '`the` *TY2*' returns a result when applied to *D*. This is expressed in Equation 8.26.

8.4 Example

To illustrate how the rules are used by the ASF evaluator, we now give an example on how the action '`(result (copy) and result 5) then apply`' is interpreted. We shall ignore the environment in this example because it does not change through the simulation. The trace can be seen in Fig. 8.4.

Since the root of the parse tree for the action contains the `then` combinator, Equation 8.4 is the first one applied. The conditions require that *eval* is applied to the left subaction ((`result (copy) and result 5`)). Here Equation 8.6, which describes the `and` combinator, is applied and again the conditions require that the left subaction (`result (copy)`) is evaluated. Using Equation 8.2 the result is '*normal* `copy`'. Returning to Equation 8.6 we see that the first condition is satisfied, and that the variable *D*2 is bound to `copy`. Now the right subaction (`result 5`) must be evaluated, and again Equation 8.2 is used and the result is '*normal* `5`'. Equation 8.6 now gives the result '*normal* (`copy, 5`)' which is returned to Equation 8.4. Thereby the first condition in Equation 8.4 is satisfied, and the variable *D*2 is bound to '(`copy, 5`)' which is used in the evaluation of the right subaction (`apply`). Equation 8.15 is now used and here the variables *A* and *D** are bound to `copy` and `5`, so it reduces to evaluating `copy` with `5` as input data. This is handled by Equation 8.1 giving the result '*normal* `5`' which is returned to Equation 8.4, and finally the result of evaluating the whole action is '*normal* `5`'.

```
eval((result (copy) and result 5) then apply, (), {})
Try Equation 8.4
    eval(result (copy) and result 5, (), {})
    Try Equation 8.6
        eval(result (copy), (), {})
        Try Equation 8.2
            normal copy
        Success Equation 8.2
        eval(result 5, (), {})
        Try Equation 8.2
            normal 5
        Success Equation 8.2
        normal (copy, 5)
    Success Equation 8.6
    eval(apply, (copy, 5), {})
    Try Equation 8.15
        eval(copy, 5, {})
    Success Equation 8.15
    eval(copy, 5, {})
    Try Equation 8.1
        normal 5
    Success Equation 8.1
    normal 5
Success Equation 8.4
```

Figure 8.4: Example of an interpretation of an action

8.5 Future work

The interpreter can be improved in several ways. Our primary goal was to quickly implement an interpreter that could easily be interfaced with the Action Environment. It was also important that it showed better performance than the interpreter implemented by van der Storm. The Fibonacci example from the beginning of the chapter shows that this goal has been achieved. If the need for an even faster interpreter arises, we probably have to chose another implementation language. Real improvement can be gained if we implement Just-in-time compilation and other modern virtual machine techniques. But since the purpose of the interpreter is to be able to evaluate small actions generated in the Action Environment to test language specifications, we think it is more important to focus on implementing debugging facilities or trace mechanisms.

Chapter 9

Type inference for Action Notation

If we knew what it was we were doing,
it would not be called research, would it?
— Albert Einstein

This chapter is a revised version of *Type inference for the new action notation* [38].

Inferring types for actions has shown to be very useful in AS-based compiler generation. It provides the user with information about the safety of actions (i.e., will type errors cause the action to err when executed) and enables compiler generators to generate optimising compilers that performs transformations based on type annotations. There has already been put a lot of effort into this area of research, but the appearance of a new version of AN (AN-2) prompted us to improve on existing work. This chapter presents a type inference algorithm that annotates AN-2 actions with types. We solve some of the problems that have cropped up in connection with the simplification of the AN kernel, and we infer types for a bigger subset of AN compared to previous work [22, 28, 30–32, 46, 75, 93].

According to the semantics of AN [62] all actions are legal, but some of them might always terminate abruptly or fail. A type error in an action will just result in a subaction terminating abruptly, which is completely legal, and the exception can always be caught. If, for instance, the action inspect is given something that is not a memory cell, it just terminates abruptly, and the evaluation can continue at the nearest surrounding catch. We shall say that an action is *type correct* if no subactions will terminate abruptly because they are given data of an unexpected type. This implies, for instance, that the action 'give the *Type*' will always terminate normally in a type correct action. Fig. 9.1 gives some examples of actions containing type errors.

The first action is not type correct because the right-hand side of the then combinator is not given a pair of integers as it expects. In the second action the right-hand side of the action expects two values but is only given one value. The third action can terminate normally in two ways, either it performs division by zero, raises an exception, catches the exception, and gives a boolean value, or it performs division by a non-zero integer and gives the integer result. The problem is that the action does not yield data of the same type in both cases,

1. (give true and give 5) then give (the integer#1 + the integer#2)
2. give the cell then update
3. give (the integer#1 / the integer#2) catch give true
4. bind("x", true) scope give the integer bound-to "y"

Figure 9.1: Examples of actions with type errors

and therefore inferring a type for the whole action would require unions of types which our type system does not support. In the fourth action the right-hand side of the scope combinator does not receive the correct bindings. The main purpose of our type inference algorithm (TI) is not to check that an action is type correct, but only type correct actions can be annotated with types using TI, so the above examples illustrates which actions can be annotated with types.

Contrary to previous work, we shall use the word *type* instead of *sort* to emphasise that AN-2 is no longer dependent on unified algebras [55], and to be more consistent with programming language terminology.

In Section 9.1 we give an overview of related work in the area; both the work on which we build and other approaches is described. Section 9.2 is devoted to explaining TI and the types, type inference rules, and the unification operator used in the algorithm. Section 9.3 contains examples of the problems we have solved in connection with the new version of AN. The set of actions accepted by TI is described in Section 9.4. In Section 9.5 we report on how TI has been implemented. The soundness of the algorithm is discussed in Section 9.6, and a description of how we have tested TI can be found in Section 9.7. And finally we conclude and suggest future work plans in Section 9.8.

9.1 Related work

Inferring types for actions has been a research area since the beginning of the 1990's, and there has been different approaches to the problem. In this section we shall investigate the different approaches to the problem and the set of actions that can be typed in existing systems.

Even and Schmidt [32] were the first to infer types for actions. Their type system supports an ML-style type inference algorithm. The subset of AN handled by their type inference algorithm only contains actions for describing normal flow of control and scopes of bindings. Furthermore unfoldings are not allowed but abstractions which have been type annotated by the user are.

In the Cantor system Palsberg [75] chose to restrict the subset of AN handled so that the languages that can be described are monomorphic and statically typed (some of the other systems mentioned in this section only infer types for parts of an action and allows other parts to be runtime type checked). The abstractions used are annotated with types by the user and self-referential bind-

ings are not allowed (they can be used in a semantics for recursive functions). Type inference was not the main goal of Palsberg's research in connection with the Cantor system, so the type system was designed so that it could easily be implemented.

Ørbæk's type checker [93], implemented in the OASIS system, is similar to Palsberg's with one improvement: non-tail-recursive unfoldings are allowed and this means that calculating a fix-point when typing unfoldings becomes a bit more complicated.

The work described by Doh and Schmidt in [30] does not seem to be comparable to other work mentioned in this section. Their objective was to derive a type checker for a language using the semantic functions in an ASD of the language. This has the advantage that they can give the user better error-messages but is not very useful for compiler generation.

In [31] Doh and Schmidt emphasise the use of facets in type inference. Their type system is not an improvement with respect to the set of AN that can be typed compared to previous work, but they claim that their way of specifying a type system is clearer. Since abstractions and unfoldings are not allowed, it becomes impossible to describe interesting programming language features such as procedures and loops. Doh allows a bigger subset of AN in the algorithm described in [28], including both unfolding and abstractions, but his analysis of abstractions is still weak. He also combines type inference with an analysis of statically known data. This is done by using a two level type system describing compile-time types and run-time types.

Brown's type system used in the Actress system [22] is an improvement compared to previous systems. The subset of AN used includes all of AN, except the actions for describing abrupt data flow and interactive processes. His use of action types as function types from pairs of record types (data and bindings) to pairs of record types allows a very precise analysis of actions. The essential work of the algorithm lies in unifying record types.

The type system used by Lee in his Genesis system [46] is almost the same as Brown's with a few minor improvements, the most important being type inference over tuples of yielders.

It would seem that the most successful type systems for AN are based on record types. Type systems with record types is a popular research area due to its application in object oriented languages. Recent advances include systems that support record concatenation, sub-typing, and recursive types [77]. In [84] Sulzmann describes record systems supporting concatenation and extension of records, removal of fields, and field labels as first class values. Pottier [80] describes a system where both conditional constraints and row variables are used to infer types.

9.2 Overview of the type inference algorithm

The structure of our type inference algorithm (TI) resembles the one used in Genesis [46]. It uses type variables and therefore a *global mapping* from variables to types is maintained. The algorithm has three stages:

1. Use the type inference rules to annotate the action in a bottom-up traversal. Using the type inference rules generates a list of constraints.
2. Repeat the following steps as long as the list of constraints change:
   - Try to solve the constraints one by one. The constraints are solved by unifying types schemes. If a constraint is solved, it is removed from the list of constraints, and the global mapping is updated with the type variable mappings resulting from the unification.
   - For every arrow $\theta \rightarrow \tau$ in the global mapping, replace every type variable θ' in τ with τ' if the arrow $\theta' \rightarrow \tau'$ is in the global mapping. Do the same for row and record variables.
   - In the remaining set of constraints replace type variables that have been assigned a type with that type.
3. If all constraints have been solved, remove type variables from the annotated action using the global mapping, otherwise report that the action could not be annotated with a type.

Step one is an implementation of the type inference rules where meeting the premises gives rise to a recursive call on the subactions occurring in the premises. A rule without premises corresponds to an action constant (a leaf in the action AST).

There are six different constraints which will be described in later sections. Solving the constraints is done by iterating over the list of constraints while trying to solve each one of them individually. A constraint is solvable if the types it contains can be unified according to the interpretation of the constraint, and the map the unification results in does not conflict with the global map, i.e., no variables are bound in both maps to types that are not unifiable. If the constraint is solvable, the global map is updated with the new mappings, and the constraint is removed from the set of constraints.

Termination of step two is guaranteed because at least one constraint is solved in every iteration, and step two does not generate new constraints. The unification operator and the removal of variables also terminates because we do not allow recursive types (an occurrence check on type variables checks this).

9.2.1 Types

The result of running TI is an action where every subaction is annotated with action types. In the annotated action also type variables have been removed, but for use inside TI we need types containing type variables (type schemes) as presented in Fig. 9.2.

In the rest of this chapter we shall use τ to range over types (*Type*), α to range over action types (*Action*), δ, β, and Γ to range over record types (*Record*), a to range over tokens, Φ to range over field sets (*Fields*), and Ψ to range over rows (*Row*).

Our type system contains no subtype ordering between types, contrary to the type system presented in Chapter 7, and this puts some limitations on the

$$
\begin{array}{lcl}
\mathit{Type} & ::= & \mathsf{integer} \mid \mathsf{boolean} \mid \mathsf{cell}(\mathit{Type}) \mid \mathsf{token}(\mathit{Token}) \mid \\
 & & \theta \mid \mathit{Record} \mid \mathit{Action} \\
\mathit{Action} & ::= & (\mathit{Record}, \mathit{Record}) \hookrightarrow (\mathit{Record}, \mathit{Record}) \\
\mathit{Record} & ::= & \mathit{Record} \wedge \mathit{Record} \mid \mathit{Fields}\ \mathit{Row} \mid \emptyset \mid \Theta \\
\mathit{Fields} & ::= & \{\mathit{Token} : \mathit{Type}, \ldots, \mathit{Token} : \mathit{Type}\} \\
\mathit{Row} & ::= & \epsilon \mid \rho
\end{array}
$$

Figure 9.2: Type Schemes

set of actions accepted by TI. This is because implementations of type inference on type systems with record types and subtypes does not run in polynomial time [77].

Every action is annotated with an action type $((\delta, \beta) \hookrightarrow (\delta', \delta'_a))$, a function type from data tuples (δ) and bindings (β) to a pair of data tuples. The record types δ' and δ'_a represents the type of data produced by the action in the case where it terminates normally (δ') and abruptly (δ'_a). We are not representing storage in action types which means that TI cannot determine if a cell given to inspect exists. Previous work [22, 46] have used a similar action type but there the last record type was interpreted as the type of the bindings produced by an action. Actions no longer produce both data and bindings. In his PhD dissertation [22] Brown suggested an action type that also described the data produced in case of abrupt termination, but he used a simpler type in the Actress system because it did not allow abruptly terminating actions.

Both the types of data tuples (product types) and bindings are represented by record types; the only difference is that the identifiers in the former are numbers. This makes the type system simpler both with respect to the number of types and the definition of type operators, like the unification operator. A record type can be a concatenation of two record types (used when giving a type to the and, and-then, and and-catch combinators) or it can be a set of token and type pairs (field sets) together with a row variable. Row variables can be instantiated with record types. Because some actions has types where a type inference rule can only describe parts of the record types, row variables are needed. To illustrate this the action 'give #1' expects data of type '$\{1 : \theta\}\rho$' where the row variable ρ means that more fields are allowed in the record type (the action can be given more data then it uses). When no more fields are allowed in a record type the row variable ϵ is used. A record type can also be the empty record type $\emptyset$. If an action has the type '$(\{1 : \mathit{integer}\}, \{\}\rho) \hookrightarrow (\{1 : \mathit{boolean}\}, \emptyset)$', it indicates that the action can only terminate normally ($\emptyset$ can also be used to indicate that the action cannot terminate normally). If $\emptyset$ is used in a concatenation of two record types, the result is $\emptyset$, i.e. '$\Gamma \wedge \emptyset = \emptyset \wedge \Gamma = \emptyset$'. Finally, a record type can be a variable Θ which can be instantiated with all record types. In previous work only the row variables where needed, and not the record variables, but the $\emptyset$ record type means that the type $\{\}\rho$, with the proper instantiation of ρ, cannot describe all record types. The alternative, only having record variables, would imply that the type given to an action like

'give #1' would not be very precise which makes it more difficult to infer a type for the action. Previous work also included three different kinds of fields, *Type*, Δ, and absent where Δ was a variable ranging over fields and absent indicated that the field should be removed from the record type. These extra fields were needed to describe some action combinators that are not part of the AN-2 kernel. An extra row variable, γ, was also used in previous work to distinguish between actions that ignore the given data and actions that use the given data. This information was used when transforming actions, but we are not going to transform actions so we have removed the extra row variable.

Data in an action can have the types derived from the nonterminal *Type*. The types are atomic types, like integer and boolean, and the constructed type cell(τ) which is the type of a memory cell holding data of type τ. A token a have the type token(a). The type of a token is parametrised by the token because the token value is needed to infer a type for data operators operating on bindings. Record types are included among the regular types which is useful when an action produces bindings as output (e.g., copy bindings, Rule 9.18). Also the action type is a regular type because it is allowed (and very useful) for an action to give an action as output.

Among the data types we also find type variables (θ), they are used in the type inference rules for create (Rule 9.23), update (Rule 9.23), and inspect (Rule 9.23).

Compared to previous work, we have added some more record types, changed the action type, and removed some field types, a row variable, and individuals as types. AN-2 is not based on unified algebras, so there is no need to look at individuals as types, and moreover this is very uncommon in type systems, so we have decided to remove them, except in connection with tokens.

9.2.2 Type inference rules

Many of the type inference rules in our system are the same as the ones used in Genesis [46] modulo the change of names, e.g., hence has changed to scope, and a change of type schemes. We have added and removed some rules because we are working on AN-2 and are looking at a bigger subset of AN.

Figs. 9.3–9.7 shows the type inference rules. In this section we will explain the most interesting rules. The rules are conditional, and to get the basic understanding of how the rules should be read, let us take a look at Rule 9.6. This rule says that if A_1 has a type and A_2 has a type, then 'A_1 and A_2' has a type. The type of data (bindings) given to the two subactions should be the same as the type of data (bindings) given to the whole action, and the resulting type of data is the concatenation of the output of the subactions ($\oplus$ is a concatenation operator on record schemes). Their output in case of normal termination should also have the same type. The action type α_u before the turnstile is only used in connection with the unfolding action (see below) but must be propagated to the subactions.

The rule for give o (o is a data operator) illustrates how some actions have types that indicate that they might both terminate normally and abruptly. If the data operator is partial (Rule 9.2), indicated by the special arrow '$\rightarrow$?', and

$$\frac{\alpha_u \vdash \mathsf{d} : \delta}{\alpha_u \vdash \mathsf{result\ d} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\delta, \emptyset)} \tag{9.1}$$

$$\frac{\mathsf{o} : \delta \rightarrow ?\ \tau}{\alpha_u \vdash \mathsf{give\ o} : (\delta, \{\}\rho) \hookrightarrow (\{1 : \tau\}\epsilon, \{\}\epsilon)} \tag{9.2}$$

$$\frac{\mathsf{o} : \delta \rightarrow \tau}{\alpha_u \vdash \mathsf{give\ o} : (\delta, \{\}\rho) \hookrightarrow (\{1 : \tau\}\epsilon, \emptyset)} \tag{9.3}$$

$$\alpha_u \vdash \mathsf{copy} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\{\}\rho_1, \emptyset) \tag{9.4}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a) \\ \alpha_u \vdash A_2 : (\delta', \beta) \hookrightarrow (\delta'', \delta'_a)\end{array}}{\alpha_u \vdash A_1\ \mathsf{then}\ A_2 : (\delta, \beta) \hookrightarrow (\delta'', \delta'_a)} \tag{9.5}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta'_1, \delta'_a) \\ \alpha_u \vdash A_2 : (\delta, \beta) \hookrightarrow (\delta'_2, \delta'_a) \\ \delta' = \delta'_1 \oplus \delta'_2\end{array}}{\alpha_u \vdash A_1\ \mathsf{and}\ A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)} \tag{9.6}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta'_1, \delta'_a) \\ \alpha_u \vdash A_2 : (\delta, \beta) \hookrightarrow (\delta'_2, \delta'_a) \\ \delta' = \delta'_1 \oplus \delta'_2\end{array}}{\alpha_u \vdash A_1\ \mathsf{and\text{-}then}\ A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)} \tag{9.7}$$

$$\frac{\alpha_u \vdash A : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)}{\alpha_u \vdash \mathsf{indivisibly}\ A : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)} \tag{9.8}$$

$$\frac{(\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\Theta_1, \Theta_2) \vdash A : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\Theta_1, \Theta_2)}{\alpha_u \vdash \mathsf{unfolding}\ A : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\Theta_1, \Theta_2)} \tag{9.9}$$

$$\alpha_u \vdash \mathsf{unfold} : \alpha_u \tag{9.10}$$

$$\alpha_u \vdash \mathsf{choose\text{-}nat} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\{1 : \mathsf{integer}\}\epsilon, \emptyset) \tag{9.11}$$

Figure 9.3: Normal flow of control and data

the data operator is not defined on the given data, the action will terminate abruptly with no data ($\{\}\epsilon$), otherwise it will terminate normally with the data computed by the data operator. Notice that the operator is expected to result

$$\alpha_u \vdash \mathsf{throw} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\emptyset, \{\}\rho_1) \tag{9.12}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a) \\ \alpha_u \vdash A_2 : (\delta'_a, \beta) \hookrightarrow (\delta', \delta''_a)\end{array}}{\alpha_u \vdash A_1 \ \mathsf{catch} \ A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta''_a)} \tag{9.13}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta', \delta'_{a1}) \\ \alpha_u \vdash A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_{a2}) \\ \delta'_a = \delta'_{a1} \oplus \delta'_{a2}\end{array}}{\alpha_u \vdash A_1 \ \mathsf{and\text{-}catch} \ A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)} \tag{9.14}$$

$$\frac{\mathsf{o} : \delta \to \mathsf{boolean}}{\alpha_u \vdash \mathsf{check} \ \mathsf{o} : (\delta, \{\}\rho) \hookrightarrow (\delta, \{\}\epsilon)} \tag{9.15}$$

$$\alpha_u \vdash \mathsf{fail} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\emptyset, \emptyset) \tag{9.16}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a) \\ \alpha_u \vdash A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)\end{array}}{\alpha_u \vdash A_1 \ \mathsf{else} \ A_2 : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)} \tag{9.17}$$

Figure 9.4: Abrupt flow of control and data

$$\alpha_u \vdash \mathsf{copy\text{-}bindings} : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\{1 : \{\}\rho_2\}\epsilon, \emptyset) \tag{9.18}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : (\delta, \beta_1) \hookrightarrow (\{1 : \beta_2\}\epsilon, \delta'_a) \\ \alpha_u \vdash A_2 : (\delta, \beta_2) \hookrightarrow (\delta'_2, \delta'_a)\end{array}}{\alpha_u \vdash A_1 \ \mathsf{scope} \ A_2 : (\delta, \beta_1) \hookrightarrow (\delta'_2, \delta'_a)} \tag{9.19}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A : (\delta, \beta_1) \hookrightarrow (\{1 : \beta_2\}\epsilon, \delta'_a) \\ \beta_1 = \beta_2/\beta_3\end{array}}{\alpha_u \vdash \mathsf{recursively} \ A : (\delta, \beta_3) \hookrightarrow (\{1 : \beta_2\}\epsilon, \delta'_a)} \tag{9.20}$$

Figure 9.5: Scopes of bindings

in a single datum type, which is translated into a record type.

In Rule 9.4 we see the use of row variables. All the type variables used in the type inference rules are fresh variables. The use of the variables ρ_1 and ρ_2 means that no restrictions are put on the type of data and bindings given to copy, except that the data produced should also have type ρ_1. We could also

$$\alpha_u \vdash \mathsf{close} : (\{1 : (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\Theta_1, \Theta_2)\}\epsilon, \{\}\rho_2) \hookrightarrow (\{1 : (\{\}\rho_1, \{\}\epsilon) \hookrightarrow (\Theta_1, \Theta_2)\}\epsilon, \emptyset) \tag{9.21}$$

$$\alpha_u \vdash \mathsf{apply} : (\{1 : (\{\}\rho_1, \{\}\epsilon) \hookrightarrow (\Theta_1, \Theta_2)\}\rho_1, \{\}\rho_2) \hookrightarrow (\Theta_1, \Theta_2) \tag{9.22}$$

Figure 9.6: Actions as data

$$\frac{\theta : \mathsf{storable}}{\alpha_u \vdash \mathsf{create} : (\{1 : \theta\}\epsilon, \{\}\rho) \hookrightarrow (\{1 : \mathsf{cell}(\theta)\}\epsilon, \emptyset)} \tag{9.23}$$

$$\frac{\theta : \mathsf{storable}}{\alpha_u \vdash \mathsf{update} : (\{1 : \mathsf{cell}(\theta), 2 : \theta\}\epsilon, \{\}\rho) \hookrightarrow (\{\}\epsilon, \emptyset)} \tag{9.24}$$

$$\frac{\theta : \mathsf{storable}}{\alpha_u \vdash \mathsf{inspect} : (\{1 : \mathsf{cell}(\theta)\}\epsilon, \{\}\rho) \hookrightarrow (\{1 : \theta\}\epsilon, \emptyset)} \tag{9.25}$$

Figure 9.7: Storage manipulation

$\oplus : \mathit{Record} \times \mathit{Record} \to \mathit{Record}$

$\emptyset \oplus \Gamma = \emptyset$

$\Gamma \oplus \emptyset = \emptyset$

$\Gamma_1 \oplus \Gamma_2 = \Theta$ **where** the constraint $C_=(\Gamma_1 \wedge \Gamma_2, \Theta)$ must be satisfied.

$/ : \mathit{Record} \times \mathit{Record} \to \mathit{Record}$

$\Phi_1\ \epsilon\ /\ \Phi_2\ \epsilon = \Phi_3\ \epsilon$ **where** Φ_3 contains $\Phi_1$and bindings from Φ_2 where the tokens are not bound in Φ_2

$\Gamma_1\ /\ \Gamma_2 = \{\}\rho$ **where** the constraint $C_/(\Gamma_1, \Gamma_2, \{\}\rho)$ must be satisfied.

Figure 9.8: Auxiliary type operations

use a record variable Θ here, but because copy always terminates normally ρ_1 is more precise. Terminating abruptly is not a possibility for copy, therefore the record type $\emptyset$ is in the resulting type.

The rules for unfolding and unfold illustrate the use of the action type before the turnstile. When using Rule 9.9 it is necessary to guess the type of the whole action 'unfolding A', which is then passed to the unfold actions contained in A. This ensures that the type of 'unfolding A', A, and any unfold contained in A has the same type. The guessing is done by constructing an action type containing row and record variables. Notice the use of row variables in the input because we know that the action type cannot contain $\emptyset$ on these two places as it can in the output. The value of α_u outside the body of an unfolding does not need to

be defined because we do not allow occurrences of unfold without an enclosing 'unfolding A'.

The type inference rules for the actions that describe abrupt control flow (Rules 9.12, 9.13, and 9.14) resembles the Rules 9.4, 9.5, and 9.7 due to the similarity in their behaviour as already pointed out in Section 2.3.2.

In Fig. 9.5 the rules for actions used to describe scopes of bindings are displayed. Looking at the rule for scope (Rule 9.19), we notice that we can also put restrictions on the types of the subactions. Previous rules have just stated that the subactions should have a type, and that this type should perhaps contain record types that also occur in the other subactions type. Here the restriction is that the left subaction produce bindings. Rule 9.20 is also interesting because the type of bindings given to the subaction A depends on the type of bindings computed by it. The intention is that A computes recursive bindings, so the bindings computed by A should also be available in the evaluation of A, which is ensured by letting the computed bindings β_2 override the bindings β_3 given to the whole action, and the result β_1 is then the bindings given to A.

The two actions that describes the reflective part of AN are displayed in Fig. 9.6. These type rules also make extensive use of row and record variables to describe that the input can be arbitrary actions (possible with some restrictions regarding the bindings used by the action). The record type variables are only used in the output of action types because the input can never be $\emptyset$.

The last type rules (Rules 9.23–9.25) all use type variables (θ) to describe that the action produce or operate on cells containing data of arbitrary type. The condition 'θ : storable' states that the type variable θ must be instantiated to a storable type. This will result in a type projection constraint ($C_{\subseteq}$) in the implementation of the rules.

The type operator $\oplus$ is defined in Fig. 9.8. Given two record types it returns $\emptyset$ if one of them is $\emptyset$, or it generates a constraint and returns a fresh record type variable. Applying the operator to $\emptyset$ is, for instance, done when A_1 or A_2 in 'A_1 and A_2' does not terminate normally, and in this case the whole action should not terminate normally, and therefore the result should be $\emptyset$. The constraint $C_{=}(\Gamma_1 \wedge \Gamma_2, \Theta)$ require that the record type $\Gamma_1 \wedge \Gamma_2$ (the concatenation of Γ_1 and Γ_2) can be unified with the record type assigned to the variable Θ.

In Fig. 9.8 also the type operator / (used in Rule 9.20) is defined. Given two record types with ϵ rows, it computes the result of letting the bindings represented by Φ_1 override the bindings represented by Φ_2. For all other record types a constraint $C_{/}(\Gamma_1, \Gamma_2, \{\}\rho)$ is generated. More about this constraint in Section 9.3.1.

9.2.3 Type inference rules for data operators

Inferring types for some data operators is more complicated then just stating the signature of the operator. Type inference rules for some of these data operators are presented in Fig. 9.9.

The intention of the type projection operator 'the τ' is that the type of data given to it should be a subtype of τ, so we cannot just give it the type '$\tau \to \tau$'. Rule 9.26 illustrates how a type variable θ is used together with the condition

'$\theta : \tau$' to describe the type of the data operator. As mentioned in the previous section the condition '$\theta : \tau$' is translated to the projection constraint $C_{\subseteq}$.

Rules 9.27–9.30 describe the data operations operating on bindings. The rule for overriding is further explained in Section 9.3.1.

$$\frac{\theta : \tau}{\mathsf{the}\ \tau : \theta \to \theta} \tag{9.26}$$

$$\frac{C_{\mathrm{bind}}(\theta_1, \theta_2, \{\}\rho); \quad \theta_2 : \mathsf{bindable}}{\mathsf{binding} : (\theta_1, \theta_2) \to \{\}\rho} \tag{9.27}$$

$$\frac{C_{\mathrm{bound}}(\{\}\rho, \theta_1, \theta_2)}{\mathsf{bound} : (\{\}\rho, \theta_1) \to \theta_2} \tag{9.28}$$

$$\frac{C_{/}(\{\}\rho_1, \{\}\rho_2, \{\}\rho_3)}{\mathsf{overriding} : (\{\}\rho_1, \{\}\rho_2) \to \{\}\rho_3} \tag{9.29}$$

$$\frac{C_{\dot\cup}(\{\}\rho_1, \{\}\rho_2, \{\}\rho_3)}{\mathsf{disj\text{-}union} : (\{\}\rho_1, \{\}\rho_2) \to \{\}\rho_3} \tag{9.30}$$

Figure 9.9: Type inference rules for data operations

Inferring types for the data operators defined in the ASDF modules involves new constructed types and the projection constraint $C_{\subseteq}$. We omit the technical details here.

9.2.4 Constraints

The implementation of the type inference rules generates a list of constraints stating properties about the types in the rules. For instance Rule 9.5 produces the constraints: input bindings for A_1 and A_2 should be the same, normal output from A_1 is equal to the input data for A_2, and the abrupt output from A_1 and A_2 should be the same (except if one of them is $\emptyset$). Formally these constraints are expressed using the $C_{=}$ constraint in two first cases and the $C_{\simeq}$ constraint in the last case. This means that TI does not accept "dead code"; if an action does not terminate normally and is followed by the then combinator, then the action is rejected by TI. If the conditions in a type inference rule state that two types should be the same, and the same type is also used in the conclusion, then the type used in the conclusion should be chosen by the following strategy: If one type is $\emptyset$, choose the other type, otherwise choose the type containing most information about the type, i.e., types without type variables are chosen over types containing type variables, and finally if a decision has not been made choose the first type.

Compared to previous work we collect more constraints to solve in the second stage of the algorithm. The constraints are listed in Fig. 9.10 where the

right column indicates what should hold to solve a constraint. The result of solving a constraint is either the result of the unification performed or, when solving $C_{\subseteq}$ (and in some cases $C_{\simeq}$), the empty map.

$C_{=}(\tau_1, \tau_2)$	:	$\mathit{unify}(\tau_1, \tau_2) \neq \mathsf{errormap}$
$C_{\simeq}(\Gamma_1, \Gamma_2)$	:	$\forall\Theta.\ (\Gamma_1 \neq \Theta \wedge \Gamma_2 \neq \Theta) \wedge$
		$(\Gamma_1 = \emptyset \vee \Gamma_2 = \emptyset \vee \mathit{unify}(\Gamma_1, \Gamma_2) \neq \mathsf{errormap})$
$C_{\mathrm{bind}}(\tau_1, \tau_2, \beta)$	:	$\tau_1 = \mathsf{token}(a)\ \wedge\ \mathit{unify}(\{a : \tau_2\}\epsilon, \beta) \neq \mathsf{errormap}$
$C_{\mathrm{bound}}(\beta, \tau_1, \tau_2)$	:	$\tau_1 = \mathsf{token}(a)\ \wedge\ \mathit{unify}(\{a : \tau_2\}\rho, \beta) \neq \mathsf{errormap}$
$C_{/}(\beta_1, \beta_2, \beta_3)$	:	$\mathit{unify}(\beta_2/\beta_1, \beta_3) \neq \mathsf{errormap}$
$C_{\dot\cup}(\beta_1, \beta_2, \beta_3)$	:	$\mathit{unify}(\beta_2\ \dot\cup\ \beta_1, \beta_3) \neq \mathsf{errormap}$
$C_{\subseteq}(\tau_1, \tau_2)$	:	$\tau_1 \subseteq \tau_2$

Figure 9.10: Constraints

The constraint $C_{\simeq}$ is used to ensure that two branches of an action results in data of the same type if only one of them terminates normally or abruptly. We cannot use $C_{=}$ here because this constraint requires that the two types are identical, whereas $C_{\simeq}$ allows one of them to be $\emptyset$ and the other something different (as it is the case if only one of the subactions combined by, for instance, the and combinator can terminate abruptly). Furthermore $C_{\simeq}$ also ensures that a type is not assigned to record variables on the top level in unification, and this is correct behaviour in this case because it is not given that the record variable should be assigned to the other type in, for instance, $C_{\simeq}(\Theta, \Phi\Psi)$; it might also be that Θ should be assigned to $\emptyset$.

The row variable ρ in third line is a fresh row variable. The type operator $\dot\cup$ in the second last line is a partial operator that forms the union of two binding maps if no tokens are bound in both maps (disjoint union).

9.2.5 Unification

To solve constraints unification is used. The unification operator *unify* takes two type schemes as arguments and returns a mapping from variables to types. Our unification operator is comparable to the one found in [46] with the main difference that ours does not return a type, and is applied after the traversal of the action instead of during the traversal.

The unification operator can be seen in Figs. 9.11 and 9.12. Unification of record types employs other type operators: *unifyfs* (see Fig. 9.13) performs unification on field sets. '$-$' takes two field sets and removes the tokens occurring in second argument from the first argument. '$\cdot$' takes two field sets representing product types and computes field sets representing the concatenation of the product types (see Fig. 9.14). *renumber* takes a number n and a field set, and if the field set represents a product type, it returns the field set with the tokens renumbered starting from n, otherwise it just returns the field set unchanged. *subset* takes two numbers, n_1 and n_2, and a field set representing a product type, and returns the fields in the range n_1 - n_2 renumbered from 1. $|\cdot|$ takes a field set and returns the size of it.

The mappings from variables to types can either be a union of arrows from variables to types or the constant map errormap which indicates that unification was not possible. The union of errormap with a set of arrows is errormap.

In a concatenation record type we can safely assume that the two record types have numbers as tokens. Furthermore we shall assume that the record types are sorted in ascending order using the tokens as keys (the sorting is lexicographic if the tokens are not numbers).

$$
\begin{array}{l}
\mathit{unify} : \mathit{Type} \times \mathit{Type} \to \mathit{Map} \\
\\
\mathit{unify}(\tau, \tau) = [] \\
\\
\mathit{unify}(\tau, \theta) = [\theta \mapsto \tau] \\
\mathit{unify}(\theta, \tau) = [\theta \mapsto \tau] \\
\\
\mathit{unify}(\mathsf{cell}(\tau_1), \mathsf{cell}(\tau_2)) = \mathit{unify}(\tau_1, \tau_2) \\
\\
\mathit{unify}(\Gamma, \Theta) = [\Theta \mapsto \Gamma] \\
\mathit{unify}(\Theta, \Gamma) = [\Theta \mapsto \Gamma] \\
\\
\mathit{unify}(\Phi_1\epsilon, \Phi_2\epsilon) = \mathit{unifyfs}(\Phi_1, \Phi_2) \\
\\
\mathit{unify}(\Phi_1\rho, \Phi_2\Psi) = \\
\quad [\rho \mapsto \mathit{renumber}(\Phi_2 - \Phi_1)\Psi] \cup \mathit{unifyfs}(\Phi_1, \Phi_2 - (\Phi_2 - \Phi_1)) \\
\quad \text{when } \Phi_1 - \Phi_2 = \{\} \\
\\
\mathit{unify}(\Phi_1\Psi, \Phi_2\rho) = \mathit{unify}(\Phi_2\rho, \Phi_1\Psi) \\
\quad \text{when } \Phi_2 - \Phi_1 = \{\} \\
\\
\mathit{unify}(\Phi_1\rho_1, \Phi_2\rho_2) = \\
\quad [\rho_1 \mapsto (\Phi_2 - \Phi_1)\rho_3,\ \rho_2 \mapsto (\Phi_1 - \Phi_2)\rho_3] \cup \\
\quad \mathit{unifyfs}(\Phi_1 - (\Phi_1 - \Phi_2), \Phi_2 - (\Phi_2 - \Phi_1))
\end{array}
$$

Figure 9.11: Unify (continued in Fig. 9.12)

When replacing the row variable ρ in $\Phi_1\rho$ with the record type $\Phi_2\Psi$, the result is either $(\Phi_1 \cdot \Phi_2)\Psi$ (concatenation of Φ_1 and Φ_2, see Fig. 9.14) if Φ_1 and Φ_2 are numbered field sets (the records represent product types), or it is $(\Phi_1 \cup \Phi_2)\Psi$. This special way of replacing row variables is needed because field sets that represents product types always have number tokens starting from '1', and if we just compute the union of such two field sets the result no longer represent a product type (for instance, it has two number '1' tokens).

9.2.6 The ML type inference algorithm

An easy way to infer types for actions might be to translate an action into an ML expression and then let the ML type inference algorithm do the job. Let us look at an example:

bind(y, 4) scope ((copy bindings and give x) then give bound)

$$unify(\Phi_1\epsilon \wedge \Phi_2\Psi, \Gamma) = unify((\Phi_1 \cdot \Phi_2)\Psi, \Gamma)$$

$$unify(\Phi\Psi \wedge \{\}\epsilon, \Gamma) = unify(\Phi\Psi, \Gamma)$$

$$\begin{aligned} &unify(\Phi_1\rho \wedge \Phi_2\epsilon, \Phi_3\epsilon) = \\ &\quad unify(\Phi_1\rho, subset(1, |\Phi_3| - |\Phi_2|, \Phi_3)\epsilon) \cup \\ &\quad unifyfs(\Phi_2, subset(|\Phi_2| + 1, |\Phi_3|, \Phi_3)) \quad \text{when} \quad |\Phi_1| + |\Phi_2| \leq |\Phi_3| \end{aligned}$$

$$\begin{aligned} &unify(\Phi_1\rho_1 \wedge \Phi_2\rho_2, \Phi_3\epsilon) = \\ &\quad [\rho_1 \mapsto \{\}\epsilon, \rho_2 \mapsto \{\}\epsilon] \cup unifyfs(\Phi_1 \cdot \Phi_2, \Phi_3) \\ &\quad \text{when} \quad |\Phi_1| + |\Phi_2| = |\Phi_3| \end{aligned}$$

$$\begin{aligned} &unify(\Theta \wedge \Phi_1\epsilon, \Phi_2\epsilon) = \\ &\quad [\Theta \mapsto subset(1, |\Phi_2| - |\Phi_1|, \Phi_2)\epsilon] \cup \\ &\quad unifyfs(\Phi_1, subset(|\Phi_1| + 1, |\Phi_2|, \Phi_2)) \quad \text{when} \quad |\Phi_1| \leq |\Phi_2| \end{aligned}$$

$$\begin{aligned} &unify(\Phi_1\epsilon \wedge \Theta, \Phi_2\epsilon) = \\ &\quad [\Theta \mapsto subset(|\Phi_1| + 1, |\Phi_2|, \Phi_2)\epsilon] \cup \\ &\quad unifyfs(\Phi_1, subset(1, |\Phi_1|, \Phi_2)) \quad \text{when} \quad |\Phi_1| \leq |\Phi_2| \end{aligned}$$

$$unify(\Gamma_1, \Gamma_2 \wedge \Gamma_3) = unify(\Gamma_2 \wedge \Gamma_3, \Gamma_1)$$

$$unify((\tau_{11}, \tau_{12}) \hookrightarrow (\tau_{13}, \tau_{14}), (\tau_{21}, \tau_{22}) \hookrightarrow (\tau_{23}, \tau_{24})) = \bigcup_{i=1}^{4} unify(\tau_{1i}, \tau_{2i})$$

$$unify(\tau_1, \tau_2) = \mathsf{errormap}$$

Figure 9.12: Unify (continued from Fig. 9.11)

$$unifyfs(\{a_1 : \tau_1, \ldots, a_n : \tau_n\}, \{a_1 : \tau'_1, \ldots, a_n : \tau'_n\}) = \bigcup_{i=1}^{n} unify(\tau_i, \tau'_i)$$

$$unifyfs(\Phi_1, \Phi_2) = \mathsf{errormap}$$

Figure 9.13: Unify fields

$$\begin{aligned} &\{1 : \tau_1, \ldots, m : \tau_m\} \cdot \{1 : \tau'_1, \ldots, n : \tau'_n\} = \\ &\quad \{1 : \tau_1, \ldots, m : \tau_n, m + 1 : \tau'_1, \ldots, m + n : \tau'_n\} \end{aligned}$$

Figure 9.14: Record concatenation

This should not type check since the right-hand side of the scope combinator does not receive any bindings of x to anything. Under the assumption that actions are translated into functions from data and bindings to data, we see two possible ways of representing the bindings used by actions: Either Bindings are represented as a dictionary data structure with strings as keys (a mapping from strings to data), or it is represented as ML records. The first one is easy to implement but does not lead to ML code the ML type inference algorithm will complain about, as it should because the action has a type error. The

type inference algorithm does not check whether a string is in the domain of a mapping from strings to data. This approach is also a very inefficient representation of bindings, and an optimising ML compiler can hardly eliminate the bindings from the code (more about this in Section 10.3). Using the second approach we would have to translate an action like 'give bound' into '`# x`', where `x` is the token the action looks up. Generating this code requires an analysis of the action so that the token given to 'give bound' is known when generating code. Such an analysis would be just as complicated as type inference because we need to know the type of all bindings given to all subactions in an action. Other examples can be presented that illustrates how type information is needed to be able to generate ML code that is type correct only if the action is type correct.

9.3 Problems with AN-2

The first proposed version of AN-2 [45] created new challenges in inferring types. The problem was that some of the abbreviations often expanded into actions which were difficult to infer a type for. This has lead to a revised version of AN-2 with an extended kernel that incorporated some of the problematic abbreviations. In this section we will take a look at the abbreviations causing problems, mention which actions have been moved back into the kernel, and describe how we solved some of the problems.

9.3.1 Scopes of bindings

All abbreviations used to describe scopes of bindings are defined in terms of kernel actions involving combinators for describing normal flow of control and data, scope, copy bindings and various data operations. Fig. 9.15 shows how 'furthermore A' is expanded to kernel notation.

furthermore A = (copy-bindings and A) then give overriding

Figure 9.15: furthermore expanded to kernel actions

TI only works on kernel actions, so we apply it to the action on the right-hand side of the equation. In previous work there was just a single type inference rule for 'furthermore A', but now TI has to combine three rules. This is illustrated in Fig. 9.16 (notice that the type inference rules have been simplified in this example; abrupt termination is not considered and ϵ after field sets is omitted).

For TI to infer the correct type, δ_2 must be the type of a tuple of two records, and τ must be the type of the record which is the result of overriding the first record with the second. Since TI starts at the leafs of the action AST, it cannot know anything about δ_2 because it depends on the type of copy-bindings and A_2. Therefore TI must "postpone" the type assignment to give overriding, and it does so by demanding the constraint in Fig. 9.17 to be satisfied. By postponing

$$\dfrac{\dfrac{\vdash \textsf{copy-bindings} : (\delta, \beta) \hookrightarrow \{1 : \beta\} \quad \vdash A : (\delta, \beta) \hookrightarrow \delta_1 \quad \delta_2 = \{1 : \beta\} \oplus \delta_1}{\vdash \textsf{copy-bindings and } A : (\delta, \beta) \hookrightarrow \delta_2} \quad \dfrac{\vdash \textsf{overriding} : \delta_2 \to \tau}{\vdash \textsf{give overriding} : (\delta_2, \beta) \hookrightarrow \{1 : \tau\}}}{\vdash \textsf{copy-bindings and } A \textsf{ then give overriding} : (\delta, \beta) \hookrightarrow \{1 : \tau\}}$$

Figure 9.16: Proof tree for typing the expansion of furthermore A

the handling of overriding, hopefully, record types have been assigned to the row variables.

$$\dfrac{\dfrac{C_/(\{\}\rho_1, \{\}\rho_2, \{\}\rho_3)}{\vdash \textsf{overriding} : \{1 : \{\}\rho_1, 2 : \{\}\rho_2\} \to \{\}\rho_3}}{\vdash \textsf{give overriding} : (\{1 : \{\}\rho_1, 2 : \{\}\rho_2\}, \{\}\rho_4) \hookrightarrow \{1 : \{\}\rho_3\}}$$

Figure 9.17: Typing overriding

Similar things have to be done for the data operations binding, bound, and disjoint-union that occur in the expansions of the other yielders and actions used to describe scopes of bindings. This should justify the type inference rules presented in Fig. 9.9.

9.3.2 Actions as data

AN-2 allows action combinators to be used as data operations from actions to actions, and this causes some problems similar to the ones described in the previous section. Furthermore the solution seems to be the same. The following action illustrates how then can be used as a data operation.

(give the datum#1 then give result_) and (give the action#2) then give _then_

When TI tries to annotate give _then_ with a type, the same problem as we saw with the data operations on bindings arises: TI does not have enough information at this point to assign a meaningful type to this action. Again we can postpone the annotation by introducing a constraint (see fig. 9.18).

$$\dfrac{\begin{array}{c} C_{\text{ac}}(\theta_1, \theta_2, \theta_3, \textsf{then}) \\ \vdash \textsf{\_then\_} : \{1 : \theta_1, 2 : \theta_2\} \to \theta_3 \end{array}}{\vdash \textsf{give \_then\_} : (\{1 : \theta_1, 2 : \theta_2\}, \{\}\rho) \hookrightarrow \{1 : \theta_3\}}$$

where

$$C_{\text{ac}}(\tau_1, \tau_2, \tau_3, iac) \;\Leftrightarrow\; \exists A_1, A_2 . \tau_i \in TI(A_i) \wedge \tau_3 \in TI(A_1 \; iac \; A_2)$$

Figure 9.18: Typing infix action combinators

($TI(A)$ means the type assigned to A by our typing algorithm TI, and *iac* is a variable ranging over infix action combinators). Similar things should be done for result_ and prefix-actions.

Since it is possible to describe many languages without using action combinators as data operations, and the abbreviations that used this feature have been moved to the kernel, we have chosen not to implement support for them in the latest version of TI.

9.3.3 Recursion and iteration

In the early version of AN-2 the actions for describing recursion (recursively A) and iteration ('unfolding A' and unfold), was not part of the kernel. When expanding them to kernel notation, actions was bound to special tokens, and this lead to recursive bindings. Let us look at the expansion of the unfold action (Fig. 9.19). A pseudo parse tree of the interesting part (boxed in Fig. 9.19) with some simplified type annotations is shown in Fig. 9.20 (notice that we have made the same simplifications as we did with the example in Fig. 9.16). The reader can convince himself about the correctness of the simplified type annotations by following a simple argument: The output of 'copy-bindings then give result_' is an action giving bindings that binds the token UNF[1] to another action because the current bindings must contain a binding of UNF to an action, otherwise we could not type check 'give the action bound to UNF'. The type of the action with the and combinator at the top has the same input as the two subactions, and as output a concatenation of the two subactions output. Since the type inference rule for the scope combinator dictates that the output data of the left action should contain bindings which are unifiable with the bindings used by the right action, we have to do the following unification

$$unify(\{\mathsf{UNF} : (\delta, \beta) \rightarrow \delta'\}, \beta)$$

We have to make β "equal" to some construct containing β using the right substitution of type variables. This is only possible if β is a recursive type and $\{unf : (\tau, \beta) \rightarrow \tau'\}$ is an unfolding of this recursive type (see Table 16 in [24]). A similar problem arises in connection with the expansion of 'recursively A'. Therefore an AN kernel without 'recursively A', 'unfolding A', and unfold would require recursive types in the type system. In the latest version of AN recursively, unfolding, and unfold have been moved back into the kernel.

9.4 The set of actions accepted by our type inference algorithm

TI is somewhat restrictive. To illustrate the set of actions accepted by TI we investigate some ML examples. ML has let-polymorphism which means it allows the following:

[1] UNF is a special token bound to A inside the body of 'unfolding A'.

```
unfold = (give the data then give result_)
         and
           ((copy bindings then give result_)
           and
           (copy bindings and result UNF
           then give bound_ then give the action)
           then give _scope_)
         then give _then_
         then enact
```

Figure 9.19: Expansion of unfold

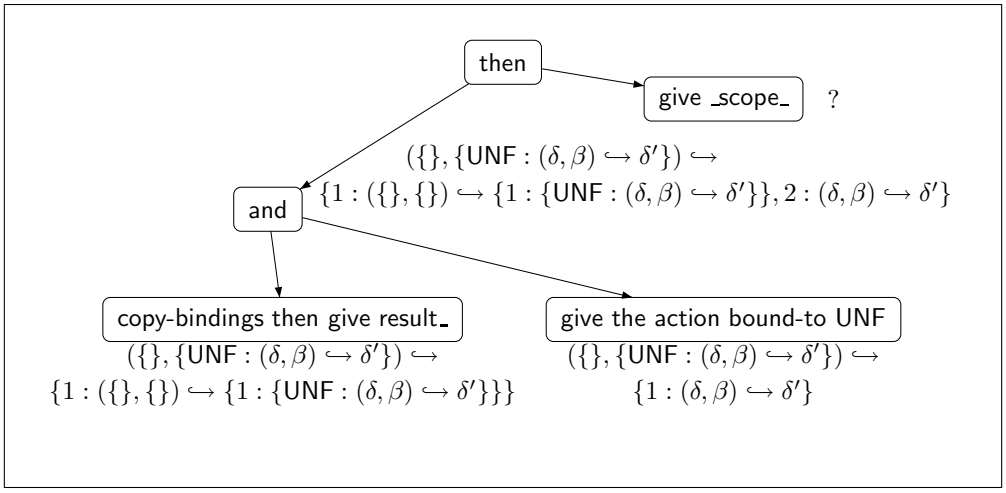

Figure 9.20: AST of the interesting part of unfold expanded

```
let
  val id = fn x => x;
in
 (id 5, id true)
end
```

The essential here is that the function `id` is polymorphic and can be used in two different contexts. Actions are also polymorphic but, unfortunately, TI does not support this. It would require extending the type schemes with universally quantified types (a second order type system [24]). The above example mapped to actions would not type check in TI.

Another example is

```
let
  exception EXC of int;
  exception ERR of bool
in
  (raise EXC 2; true) handle ERR b => b
```

```
end
```

which type checks, but raises an uncaught exception `EXC`. For the action combinator `catch`, which corresponds to ML's `handle`, TI insist that the "exception" raised in the left-hand side of the action has the same type as the data expected by the right-hand side action, and furthermore it insist that if two subactions combined by `then`, `and`, `and-then`, `else`, or `scope` can terminate abruptly, then they should result in the same type of abrupt data. We might consider relaxing these conditions, but contrary to ML, where all exceptions has the type `exn`, and the specific kind of exception is determined by the pattern right to `handle` that matches the exception, we do not have the declaration of an exception as aid when inferring the type of data given to the right-hand side subaction of `catch`; data does not have to be declared as exceptions to be raised in AN, e.g., an integer can be raised. Mapping the above example to an action will result in an action with a type error.

9.5 Implementation

TI has been implemented in ASF+SDF. Some of the pros and cons of this language has already been mentioned in Section 3.4, but we shall mention two more advantages here. The conditional type inference rules maps easily to conditional equations which makes it easy to implement them and easy to understand the implementation. As with the other tools presented in this paper it is also an advantage that interfacing to the Action Environment is easy. A disadvantage is that, as far as we are aware, it is impossible to implement arrays with constant lookup and update time in ASF which makes our implementation slow (the lookup and update time in our implementation is $O(n)$ where n is the size of the array).

The complexity of the algorithm depends on the number of constraints collected during the action traversal. The number of constraints is proportional to the size of the action. In the worst case the algorithm only solves one constraint in each iteration over the list of constraints, and then the complexity becomes $O(n^2)$ where n is the size of the action. In practise most of the constraints are solved in first attempt.

Previous work has used SML as implementation language which also has some advantages, like fast executable and a broader user community and therefore better documentation and tool support.

9.6 Soundness

In [22] Brown proved the soundness of a type inference algorithm that resembles ours. We shall not derive a similar proof here but just sketch how we think it can be done. First we must state the soundness criterion:

Definition: For an action A and a type $(\delta, \beta) \hookrightarrow (\delta', \delta'_a)$ the judgement $\vdash A : (\delta, \beta) \hookrightarrow (\delta', \delta'_a)$ is *sound* iff the following holds:

> If data d and bindings b are instances of δ and β then an performance of A will, if it terminates and does not fail, either terminate normally with data d' or abruptly with data d'_a, where d' is an instance of δ' and d_a is an instance of δ'_a.

To prove that TI is sound, we must prove that if it derives a type α for an action A, then the judgement $\vdash A : \alpha$ is sound. This can be done by structural induction over the structure of actions. For every infix action combinator *iac*, we must assume that sound type judgements exists for actions A_1 and A_2 with types α_1 and α_2 respectively. Then we must use the type inference rule and the semantics for the combinator *iac* together with the induction assumption to proof that a judgement including $A_1 iac A_2$ and a type inferred using α_1 and α_2 is sound. Similar proofs should be constructed for prefix action constructors. The action constants are the basis in the induction. For use in the structural induction, we must also prove that the type operators and unification behaves as expected.

9.7 Evaluation

The implementation has been tested on a representative set of examples based on the Core ML example (Chapter 5) and a small subset of C. In both cases TI was part of the action compiler (more about this in Section 10.4). The Core ML example has to be modified for the generated actions to type check. The problems lie in the constructs defined in the modules Exp/Val-Id-Const (Module 4 on page 51), Par/Val-Or-Var (Module 11 on page 55), and Exp/Alt-Seq (Module 9 on page 176). In the first two modules the problem is the use of the type projector 'the τ' which is intended to go wrong for the constructs to behave correctly in some cases. Since TI rejects actions that go wrong because the wrong type of data is given to a subaction, we have to use other constructs to represent identifiers in expressions and patterns. We solved the problem by distinguishing between identifiers that occurs as normal identifiers and as data constructors when constructing the BAS term and then use two constructs for each of the problematic constructs. The construct defined in the module Exp/Alt-Seq is problematic because the else is a data operator on actions which is not implemented in the latest version of TI, and again the solution was to use another module.

The tests revealed that the efficiency of the implementation can be improved; the execution time is not acceptable for examples that involve recursive functions which seems to generate actions that are complicated to type check. Furthermore the tests revealed the limitations of TI described in Section 9.7.

More exhaustive tests should be performed. The algorithm should also be run on actions coded by hand that can test some special cases.

9.8 Conclusion

We have shown how to infer types for AN-2 actions using an algorithm comparable to the ML type inference algorithm. Our work is an improvement and

adjustment of the work performed by Brown and Lee in their PhD dissertations.

Our improvements are in the following areas: First of all the algorithm now works on the new version of AN, which is not a trivial improvement due to the enhanced expressibility in AN-2. Secondly we have tried to handle a bigger subset of actions including all action combinators and constants, except the ones used to describe interactive processes.

We conclude that it is possible to infer types for a non-trivial subset of AN-2. Our algorithm has been implemented, and it has shown to be useful in practise.

There are several ways of improving TI:

- Allowing a bigger subset of AN-2 to be typeable. There are some actions which will run without problems but at the moment are not accepted by TI. Furthermore we do not handle actions used to describe interactive processes. We are only allowing statically typed actions, but it might also be interesting to be able to do something with the dynamically typed actions.

- Introduce overloaded operators. In some way we already allow overloaded operators in TI namely the built-in operators on bindings (binding, bound, overriding and disjoint-union). They are overloaded in the way that they operate on many different binding maps, and the type system regards two binding maps as different types if they do not bind the same tokens. This might give us a clue about how to implement overloaded operators in general; we could use special constraints as we did with the operators on bindings.

- Introduce subtypes. Having subtype relations between the types could give partly the same effect as overloaded operators, for instance, instead of having a plus operator on integers and one on naturals, we could have a subtype relation between the two types. As mentioned in Section 9.1, type systems that supports records and subtypes already exist, but unfortunately the type inference algorithms for these systems does not run in polynomial time [77].

- Investigate an optimal order of solving the constraints. The subtype constraint '$C_{\subseteq}$' does not contribute with any information about type variables, so it is optimal to try to solve these constraints last. Other properties of the constraints could influence the optimal order of solving the constraints.

Chapter 10

Generating code from actions

If the programmer can simulate a construct faster than a compiler can implement the construct itself, then the compiler writer has blown it badly.
— Guy Steele

Automatically generating a compiler from a formal description of a language does not always lead to efficient compilers. A formalism that supports easy construction of readable, complete, and reusable descriptions of most programming languages and at the same time has tool support for automatically generating efficient compilers seems to be non-existing. One formalism that tries to satisfy these requirements to a language description formalism and allows automatic generation of efficient compilers is AS. By efficient compilers we mean compilers that produce fast code, and not compilers that run fast or produce small code. An AS-based compiler generator produces a front end which maps each program in the described language to an action. The front end is then connected to an *action compiler*, and the result is a compiler for the described language. Previous results [93, 94] have shown that it is possible to generate compilers that produce code that is less than ten times slower than the code generated by handwritten compilers, and in some cases even as fast as only two times slower. Some restrictions have been put on the actions handled by the compiler to achieve this result, and often the implementation of the code generator in the action compiler is very complicated.

In this chapter we present an action compiler that produces more efficient code than previous action compilers, and on some examples only a factor two slower than the code produced by the Gnu C compiler. The code generator translates actions to Standard ML (SML) [51] in a straight forward way. The SML code is then compiled to an executable using the MLton[1] compiler.

An action compiler annotates and transforms the action in several steps. Our action compiler performs type inference (Chapter 9) and code generation (the main topic of this chapter), but no optimisations on the action as seen in previous results. Instead we generate code that can easily be optimised by MLton.

The rules for translating actions into SML, is described in Section 10.1. In Section 10.2 we take a look at previous work on compiling actions. Before eval-

[1] http://www.mlton.org/

uating the action compiler in Section 10.4 we take a look at the optimisations performed by MLton in Section 10.2. In Section 10.5 the limitations of our action compiler are discussed. Section 10.6 concludes.

10.1 Code generation

We have chosen to use SML as the target language for our action compiler. Previous work has used SPARC assembler code [94], C [23, 83], Java [46, 83], and a tailor made bytecode language [83] (see Section 10.2), but we found the translation from AN to SML more natural due to AN's resemblance with functional languages. The formal semantics of SML [51] should make it relatively easy to prove that the produced code is semantically equivalent to the target action.

The translation is described using conditional rules, some of which are shown in Figs. 10.1–10.5, and the rest in the appendix. An action is inductively translated to SML by translating its subactions and then combining the produced code such that it captures the semantics of the action (Rule 10.3 illustrates this). Every action is translated into an anonymous function on the form '`fn (t, b) =>` E', where `t` is the data and `b` the bindings given to the action. The expression E computes the result of applying the function, which corresponds to the data produced when evaluating the action.

In this section we will look at a representative selection of the rules; the rest can be found in the appendix. We shall use O to range over data operators, E to range over SML expressions (e.g., anonymous functions), d and I to range over SML identifiers, n to range over integers, i and j to range over labels, and t to range over types. Some rules use the function $\mathcal{T}$ that takes an action and returns its type. The type of an action is of course context-dependent and has been derived by the preceding type inference. We shall also assume that all tokens occurring in an action have been mapped into identifiers that are not reserved words in SML.

10.1.1 Flow of control and data

The action copy has the simplest translation (Rule 10.1) since it just returns the data given to it. Translating '`result` D' is only a little bit more complicated (Rule 10.2); here the data D produced by the action must be translated into an SML expression E. If the data D is an action it is translated using the rules, in other cases it is translated into SML representations of the data, e.g., integers and booleans are just translated to the same integers and booleans.

Normal composition of actions, as described by the then combinator, naturally translates to composition of the translations, E_1 and E_2, of the two subactions. The result of applying E_1 to the given data and bindings is given to E_2 together with the same bindings used by E_1. Another solution would be to preface the generated code with a function '`fun asthen (A1, A2) (t, b) = A2(A1(t, b), b)`', and then translate the action to `asthen` (E_1, E_2). A similar solution can be applied in some of the other code rules, namely the cases where the rules are not dependent on the type of the action. It will not change

$$\mathsf{copy} \longrightarrow \texttt{fn (t, b) => t} \tag{10.1}$$

$$\frac{D \longrightarrow E}{\mathsf{result}\ D \longrightarrow \texttt{fn (t, b) =>}\ E} \tag{10.2}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \end{array}}{A_1\ \mathsf{then}\ A_2 \longrightarrow\ \texttt{fn (t, b) =>}\ E_2(E_1\texttt{(t, b), b})} \tag{10.3}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ n_1 = |normout(\mathcal{T}(A_1))|,\ n_2 = |normout(\mathcal{T}(A_2))| \end{array}}{\begin{array}{l} A_1\ \mathsf{and\text{-}then}\ A_2 \longrightarrow\ \texttt{fn (t, b) =>} \\ \qquad \texttt{let val (}d_1, \ldots,\ d_{n_1}\texttt{) =}\ E_1\texttt{(t, b);} \\ \qquad\quad \texttt{val (}d_{n_1+1}, \ldots,\ d_{n_1+n_2}\texttt{) =}\ E_2\texttt{(t, b)} \\ \qquad \texttt{in (}d_1, \ldots,\ d_{n_1+n_2}\texttt{) end} \end{array}} \tag{10.4}$$

Figure 10.1: Normal flow of control and data

the execution time of the produced code, or make the translation remarkably easier, so to keep the uniformity of the code rules we have chosen the other translation.

The and-then combinator translates to a `let-in-end` expression (Rule 10.4). This expression can be used to describe declarations that are local to an expression. Both of the translations of the two subactions are evaluated on the given data and bindings, and the elements in the resulting tuples of data are bound to variables $d_1, \ldots, d_{n_1+n_2}$ which are then used in the resulting data tuple. When given an action type, the function *normout* returns the record type that describes the type of data produced by an action in case of normal termination. The $|\cdot|$ operator computes the size of the record, so a tuple pattern with the right number of d's can be generated. The cases where $normout(\mathcal{T}(A))$ is $\emptyset$ should be handled by other rules because it indicates that part of the action will never be evaluated. The rules can be found in the appendix.

$$\mathsf{unfold} \longrightarrow \texttt{fn (t, b) => UNF(t, b)} \tag{10.5}$$

$$\frac{A \longrightarrow E}{\mathsf{unfolding}\ A \longrightarrow \texttt{let val rec UNF =}\ E\ \texttt{in UNF end}} \tag{10.6}$$

Figure 10.2: Iterative control flow

The actions '**unfolding** A' and **unfold** (Rule 10.6 and 10.5 in Fig. 10.2) are used to describe iteration. The semantics of **unfold** is that it evaluates the action A in the nearest enclosing '**unfolding** A'. The translation of **unfolding**

just binds the translation of A to the identifier `UNF`, so the translation of unfold should just apply the function bound to `UNF` to the given data and bindings. The function resulting from translating unfolding is the function bound to `UNF`. Notice the use of the `rec` keyword which ensures that `UNF` can be used from within E. It is not required that the unfold call is tail recursive (as it is the case in some previous work [23]), but the SML compiler is able to optimise the code in the cases where it is tail recursive.

$$\frac{\begin{array}{c} O \longrightarrow E \\ I = excepid(\{\}) \end{array}}{\begin{array}{ll} \textsf{check}\ O \longrightarrow & \texttt{fn (t, b) =>} \\ & \quad \texttt{let val ch =}\ E\texttt{(t, b)} \\ & \quad \texttt{in if ch then t else raise}\ I\texttt{()} \\ & \quad \texttt{end} \end{array}} \tag{10.7}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I = excepid(abruptout(\mathcal{T}(A_1))) \end{array}}{\begin{array}{ll} A_1\ \textsf{catch}\ A_2 \longrightarrow & \texttt{fn (t, b) =>} \\ & \quad (E_1\texttt{(t, b) handle}\ I\ \texttt{et =>}\ E_2\texttt{(et, b)}) \end{array}} \tag{10.8}$$

Figure 10.3: Abrupt control flow

Abrupt data flow in AN is translated to raising and handling SML exceptions (Fig. 10.3). If the result of applying a data operator O to the given data is the boolean value $false$, the action '`check` O' (Rule 10.7) terminates abruptly with no data. Because SML requires that exceptions are declared before being used, some preprocessing of the whole action is needed; for every unique occurrence of a record type representing the type of data produced by a subaction that terminates abruptly a new exception is declared. The unique exception name tied to a record type is returned by the function *excepid* when it is applied to a record type. Since check does not produce any data when it terminates abruptly, the record type given to *excepid* is the empty record {}. In the generated code the SML keyword `raise` is used to raise an exception. If the result of applying the data operator (`ch`) is true, the given data (`t`) is the result.

For handling abrupt termination AN provides the action combinator catch (Rule 10.8). The code generated from it uses the SML keyword `handle` to capture the exception raised by the evaluation of the left hand side expression (E_1 `(t, b)`). The pattern 'I `et`' on the right hand side of `handle` ensures that only the right exceptions are handled, and that the raised data is bound to the identifier `et`. The alternative when the first expression terminates abruptly is to evaluate the second expression with the data raised and the original bindings. The type operator *abruptout* returns the type describing the data produced by an action in case of abrupt termination.

10.1.2 Bindings and storage

The actions concerned with scopes of bindings are shown in Fig. 10.4. The action copy-bindings (Rule 10.9) resembles the action copy and therefore the generated code is also similar. The only difference is that the result of evaluating it is the given bindings instead of the given data.

$$\text{copy-bindings} \longrightarrow \texttt{fn (t, b) => b} \tag{10.9}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \end{array}}{A_1 \text{ scope } A_2 \longrightarrow \texttt{ fn (t, b) => } E_2\texttt{(t, } E_1\texttt{(t, b))}} \tag{10.10}$$

$$\frac{\begin{array}{c} A \longrightarrow E \\ \{i_1 : t_1,\ \ldots,\ i_n : t_{n_i}\} = \mathit{bindings}(\mathcal{T}(\text{recursively } A)) \\ \{1 : \{j_1 : t_1,\ \ldots,\ j_n : t_{n_j}\}\} = \mathit{normout}(\mathcal{T}(A)) \\ B = \{j_1 \texttt{ = r1, } \ldots\texttt{, } j_{n_j} \texttt{ = r}n_j\}/\{i_1 \texttt{ = ref b1, } \ldots\texttt{, } i_{n_i} \texttt{ = ref b}n_i\} \end{array}}{\begin{array}{l} \text{recursively } A \longrightarrow \\ \texttt{fn (d, } \{i_1 \texttt{ = b1, } \ldots\texttt{, } i_{n_i} \texttt{ = b}n_i\texttt{) =>} \\ \texttt{let} \\ \quad \texttt{val rec rv1 = fn x => (rv1 x);} \\ \quad \texttt{val r1 = ref rv1;} \\ \quad \ldots \\ \quad \texttt{val rec rv}n_j \texttt{ = fn x => (rv}n_j \texttt{ x);} \\ \quad \texttt{val r}n_j \texttt{ = ref rv}n_j\texttt{;} \\ \quad \texttt{val } \{j_1 \texttt{ = ref f1, } \ldots\texttt{, } j_{n_j} \texttt{ = ref f}n_j\} \texttt{ = } E\texttt{(t, } B\texttt{)} \\ \texttt{in} \\ \quad \texttt{r1 := f1; } \ldots\texttt{; r}n_j \texttt{ := f}n_j\texttt{; } \{j_1 \texttt{ = f1, } \ldots\texttt{, } j_{n_j} \texttt{ = f}n_j\} \\ \texttt{end} \end{array}} \tag{10.11}$$

$$\text{create} \longrightarrow \texttt{fn (t, b) => ref t} \tag{10.12}$$

Figure 10.4: Scopes of bindings

The same similarity can be seen when comparing the combinators scope (Rule 10.10) and then (Rule 10.3). The second subaction A_2 in 'A_1 scope A_2' uses the bindings produced by A_1 together with the original data, and this is reflected in the way the functions generated from the subactions are composed.

More interesting is the rule for recursively (Rule 10.11), and this is also the most complicated of the code generation rules. When evaluating the action 'recursively A' the bindings produced by the subaction A is also part of the bindings given to A. The bindings b_1 given to 'recursively A' (in the rule it is $\mathit{bindings}(\mathcal{T}(\text{recursively}A))$) and the bindings b_2 produced by A are combined by letting b_2 override b_1, and the result is given to A. This allows recursive declarations, like for instance recursive functions, in A.

To capture this relatively complex semantics, we use a trick where we bind

every identifier in the domain of b_2 to a reference containing a "dummy" value, and every identifier in the domain of b_1, but not in the domain of b_2, is bound to a reference containing the value originally bound to the identifier. These bindings are then given to the code generated from A, which produces new bindings. Finally these bindings are used to update the references containing dummy values with the correct values. Using infinitely recursing functions as dummy value ensures that all functions can be stored in the reference because the reference will hold functions of type $\alpha \to \beta$, where α and β are type variables. In Rule 10.11 A is expected to generate bindings where actions are bound to identifiers, but if it binds other types of values, the dummy values used in the generated code should be changed to values with the same types as the bound values.

From the above, we see that looking up bound identifiers and creating new bindings in A must also take account of the use of references by dereferencing and creating references. Our implementation of '`recursively` A' resembles the new interpretation of '`recursively` A' suggested by Mosses as explained in Section 2.4.

There are three actions to describe manipulation of storage, but only the one for allocating new memory cells is shown in Fig. 10.4. The generated code for `create` takes advantage of the built-in SML datatype for references. The constructor `ref` is used to construct a reference containing the data given to the function. For the two other actions, `inspect` and `update`, two other SML data operations on references, `!` and `:=`, are used to lookup the value stored in a reference and store a new value in an existing reference.

10.1.3 Actions as data

The biggest advantage in using SML as target language is in the translation of the actions related to actions as data. Here we exploit the fact that SML has higher order functions. When the data produced by '`result` D' is an action, it is useful that SML allows a function to return a function as result. For the action `apply` the generated code is a function that expects a function (d_1) together with some data ($d_2, \ldots, d_n$) and then applies d_1 to the data $d_2, \ldots, d_n$ together with the empty record representing no bindings.

$$\frac{n = |normout(\mathcal{T}(\texttt{apply}))|}{\texttt{apply} \longrightarrow \texttt{fn ((}d_1\texttt{, ..., }d_n\texttt{), b) => }d_1\texttt{ ((}d_2\texttt{, ..., }d_n\texttt{), \{\})}} \quad (10.13)$$

$$\texttt{close} \longrightarrow \texttt{fn (a, b) => fn (t, \{\}) => a (t, b)} \quad (10.14)$$

Figure 10.5: Actions as data

The action `close` results in a function that both expects a function (the parameter `a`) and produces a function (`fn (t, {}) => a (t, b)`). The produced function expects no bindings and just applies the function `a` to the data and the bindings given to the whole function.

10.1.4 Data and data operators

AN contains a number of built-in data operators on integers and booleans that can trivially be translated to corresponding SML data operators. The built-in data operators on binding maps (binding, bound, overriding, disj-union) are translated into selection of elements from records and construction of records. To translate these data operators the type information about the given bindings is used. ASDF lets the user specify data and data constructors, and these are also translated into SML by the action compiler.

10.1.5 Example

To finish this section, we will give an example of the result of translating an action to SML. The action '(copy and (result 5 then create)) then apply' expects an action and then applies this action to a memory cell containing the integer 5. The translation is shown in Fig. 10.6 (we have added integer postfixes to some identifiers to improve readability, and inserted comments describing which subaction a subexpression originates from).

```
(fn (t1, b1) => (* then *)
  (fn ((d1, d2), b2) => d1 (d2, {})) (* apply *)
     ((fn (t3, b3) => (* and *)
         let val d1 = (fn (t4, b4) => t4) (t3, b3); (* copy *)
             val d2 = (fn (t5, b5) => (* then *)
                         (fn (t6, b6) => ref t6) (* create *)
                            ((fn (t7, b7) => 5) (* result 5 *)
                                (t5, b5), b5))
                         (t3, b3)
     in (d1, d2) end)
     (t1, b1), b1))
```

Figure 10.6: Example of generated code

Notice that the order of the subexpressions representing subactions is reversed compared with the whole action, when the subactions are combined using then. The `let-in-end` expression is the translation of the subaction with and as root, and here the results of evaluating its two subactions are bound to the identifiers `d1` and `d2` which are then combined into a pair; the result of the whole subaction.

10.2 Related work

The Actress system [23] showed how to compile actions into C code. The compilation involved several action optimisations where the most important one was binding elimination. The system has been tested on a specification of a small imperative language called *Specimen*, and the running times of the generated C code for some programs have been compared to running times for

implementations of the same programs in Pascal. This comparison shows that the generated C code is between a factor 5 to 28 slower than the compiled Pascal code. The rules describing the code generation are complicated because they use a set of variables to pass data between actions and must keep track of which variables are used and have been used by subactions.

Peter Ørbæk's OASIS [94] generated SPARC assembler code. This system applied several optimisations known from handwritten compilers, like constant propagation and tail recursion detection. In a comparison between programs compiled with a generated compiler for an imperative language *HypoPL* and equivalent programs written in C and compiled with GCC 2.4.3 (with full optimisation), Ørbæk showed that the code from the generated compiler was between 1.6 to 4 times slower. Due to the low level of the target language, the code generation is complicated[2].

Continuing the work done by Brown, Moura and Watt on the Actress system, Kent D. Lee developed the Genesis system [46]. The systems have many similarities with respect to type inference and action transformations, but instead of generating C code, Genesis generates Java bytecode. One advantage of this is the portability of the generated code. As with the OASIS system, the low level target language makes code generation complicated, and special transformations of actions are needed. Lee does not present any evaluation of the generated code.

A somewhat different approach has been demonstrated by Bondorf and Palsberg in [7]. By writing an action interpreter in Scheme and applying the Similix partial evaluator, they were able to generate an action compiler that generates Scheme code. The advantage of this approach is that it is easier to write an action interpreter than an action compiler, and the hard work is done by Similix. Should AN change it is also easier to update an action interpreter than an action compiler. Their evaluation of the generated scheme code shows that it is almost 100 times slower than code generated by a hand written compiler.

Recently Tijs van der Storm [83] has shown a simpler approach to compiling actions to C and Java. Comparing it with Actress and Genesis the compiler is simpler because it does not perform any type inference or optimisations. Instead of translating an action combinator to a sequence of statements, it translates it to a function that calls the functions representing the subactions. Because his compiler does not perform type inference, the compiler cannot produce code that can easily be optimised by the C (or Java) compiler which is reflected in his test results. Van der Storm has only documented a test where he uses an action calculating Fibonacci numbers [83, Section 5]. The best result achieved when calculating the 20th Fibonacci number is a running time of 0.8 seconds. To compare we have achieved a running time of 0.5 seconds for calculating the 33rd Fibonacci number on slower hardware (Intel® Pentium® III 1 Ghz) than the hardware used by van der Storm (AMD XP® 1800+). We were not able run his action compiler to better compare the two compilers.

[2]Code generation is not well documented in [94], but the source code of OASIS can be downloaded at `ftp://ftp.daimi.au.dk/pub/empl/poe/oasis-2.2.tar.gz`

There is a huge selection of compiler generators employing other formalisms than AS available. We shall mention two systems here that also seems to be popular outside academia, contrary to the AS-based systems.

The Eli system [34] is based on attribute grammars. In addition to using attribute grammars the user can specify part of the compiler by "analogy" which means that the system has a large library of constructs used in common programming languages, so if the user wants scope rules similar to the ones used by Algol 60, he should just include the right module in the specification instead of writing it from scratch. The user can also specify part of the compiler by "solution" which means that he can write arbitrary fragments of C code that solves a problem. There are no examples in the literature of using Eli for implementing compilers for functional or object oriented languages, but a large set of real world imperative languages (Algol 60, C, Pascal) have been implemented completely or partly. In [48] the compiler for a Pascal-like language generated by Eli is compared with GCC, and the results show that the Eli generated compiler produces code that is approximately 35 % slower than the code produced by GCC.

In Gentle [82] the specification of a compiler is done in a logic programming language which is used in all parts of the specification. The specification language resembles Prolog but is more restricted and therefore the unification algorithm could be optimised. In [88] Vollmer reports that Gentle generates very efficient compilers with respect to compilation time and user experience shows that developing compilers in Gentle saves time compared to hand-coding compilers.

10.3 Optimisations performed by MLton

Previous work contains a number of transformations of actions. In this section we investigate how the optimisations performed by MLton compares to these transformations. Letting MLton perform the optimisations speeded up the development and implementation time of our action compiler, but the question is whether we can improve it using action transformations?

The MLton compiler transforms the ML code into executable machine code through various intermediate languages. For every intermediate language a different set of optimisations is performed.

Some of the transformations performed by other action compilers are listed in Fig. 10.7.

The **algebraic simplifications** essentially removes subactions that do not have any effect on the final result when evaluating an action. To take an example let us look at the equality '`copy then` $A = A$', which is the basis for one of the algebraic simplifications. Translating '`copy then` A' to SML gives us

```
fn (t1, b1) => (fn (t2, b2) => t2) (A (t1, b1), b1)
```

where `A` is the translation of the action A to SML. MLton inline functions which transforms the code into ('`fn (t2, b2) => t2`' is inlined)

Algebraic simplification	Uses the algebraic properties of AN to simplify actions. As an example '`copy then` A' has the same behaviour as A.
Transient elimination	Actions can be simplified when parts of it receive data that is known on compile time.
Binding elimination	If the language is statically bound, all bindings can be eliminated from the action, meaning that all occurrences of `give bound` can be replaced with `result` d, where d is the data that should be looked up.
Static allocation	If an analysis says that a memory cell can safely be allocated on the stack, and not on the heap, `create` is replaced with `result` c where c is a statically allocated memory cell.

Figure 10.7: Action transformations

```
fn (t1, b1) => A (t1, b1)
```

when this function is applied to a value MLton will often inline it so the effect is that just `A` is applied. All in all MLton will perform the algebraic simplification. Similar arguments can be given for the other algebraic simplifications.

Transient elimination is essentially constant propagation and folding which is also performed by MLton. The constant propagation performed by MLton is global so constants can escape functions.

Binding elimination can either replace the action that looks up a binding with an action that produces constant data, or it can replace it with an action that inspects a memory cell, depending on whether the data was known when it was bound or not. Since bindings are represented by records in SML and the action that looks up bindings is translated into '`# id b`' (where `id` is the token being looked up, and b the record), the MLton compiler can translate it into inspecting part of the memory where the record bound to `b` is stored. What is crucial here is that the token being looked up is known on compile time. In some cases constant propagation in MLton can also replace '`# id b`' with the right constant.

The MLton performs various optimisations on references similar to the ones performed by previous work to ensure **Static allocation** in as many cases as possible. The idea is to check which allocated references/memory cells escape a function/action.

10.4 Evaluation of the action compiler

The action compiler has been implemented using ASF+SDF, a formalism that makes it easy to implement especially the code generation rules. The drawback

is that it does not lead to fast executables, and therefore we are not going to compare the compilation times in this section, as it is done in related work.

We have tested the action compiler as part of our compiler generator, meaning that we have given the compiler generator two descriptions of programming languages and then compiled some test programs with the generated compilers.

The tests were run on a 3.0 GHz Intel® Pentium® 4 with 512 Kb cache and 1Gb RAM running Linux 2.4.20-31.9. The generated SML code from the action compiler was compiled using the MLton 20040227 compiler.

The first language we tested was the Core ML language as described in [41]. In Table 10.1 the test results are shown. The following test programs were used:

fibo uses a recursive function to calculate the 40th Fibonacci number.

ackerman computes the Ackerman function on the integers 3 and 11.

fibo-while calculates the 40th Fibonacci number using a while loop and references. The calculation is repeated 2 million times to reduce the significance of the program startup time.

length declares a list datatype, then constructs a list of length 100000 using a recursive function, and finally calculates the length using another recursive function.

church constructs the Church encoding of 10 million, and then transforms the Church encoding of the number back to an integer by applying the encoding to the increment function and 0.

The test programs exploit both the functional and the imperative aspects of the Core ML language. The second column shows the running time for the output from the action compiler. The third column shows the running time for the program compiled with the MLton compiler, and the last column shows how many times slower the output from the generated compiler is.

Program	**Generated compiler**	**MLton**	**Factor**
fibo	4.80 s	2.77 s	1.7
ackerman	4.33 s	0.63 s	6.9
fibo-while	1.24 s	0.34 s	3.6
length	1.13 s	0.21 s	5.4
church	7.83 s	0.33 s	23.7

Table 10.1: CoreML running times

The result for the **fibo** program is quite satisfying, while the results for **ackerman**, **fibo-while**, and **length** are acceptable. The main reason for the slowness of the **fibo-while** and **length** programs is the way we represent data types (references and lists) in the produced SML code. Because of the way the

semantics of function application in Core ML is described, the action representing the **ackerman** program is not tail-recursive[3], as the ML program is, and therefore the MLton compiler does a better job in optimising the ML program than it can do on the code produced by our action compiler on this program. The problem with tail recursion is not noticeable in the recursive Fibonacci program (**fibo**), because the recursive function there is not tail recursive. In the AS description of Core ML the action representing a function is wrapped in a data type, and this is the main reason for the bad results when running the **church** test program which exploits higher-order functions.

The compiler generator has also been tested on a small subset of C called *miniC*. The subset includes simple expressions, assign-, if- and while- statements, statement blocks, variable declarations, and recursive functions. The values are integers and arrays of integers, but no pointers. The seven test programs are:

fibo computes the 40th Fibonacci number using a recursive function.

ackerman computes the Ackerman function on the integers 3 and 11.

decrement contains four mutually recursive functions calling each other while decrementing an integer argument from 10 million to zero.

fibo-while calculates the 40th Fibonacci number using a while loop. The calculation is repeated 50 million times to reduce the significance of the program startup time.

euclid is an implementation of Euclid's algorithm that finds the greatest common divisor of 37 and 1023. This is repeated 20 million times.

sieve is an implementation of the Sieve of Eratosthenes that finds the prime numbers between 1 and 2 million. This is repeated 10 times.

bubble is an implementation of the bubble-sort algorithm on an array of integers of length 32000.

Table 10.2 compares the running times for the output from the generated compiler and the programs compiled with GCC 3.3.2.

The results for all test programs except **ackerman** and **decrement** are quite satisfying. On these programs the generated compiler generates code that is only a factor two slower than code generated by GCC on the average. We think that the generated compiler produces so slow code on the **decrement** program, compared to the GCC compiler, because of the way the recursively combinator is implemented, which seems to be particularly inefficient when the program contains mutually recursive functions. In the **ackerman** and **decrement** programs the problem with non-tail recursive actions appears again.

Comparing a compiler generated from a subset of C with a compiler for the whole C language is of course not fair. It is likely that the generated compiler

[3]The else combinator used in Module 5 on page 52 is the reason why the invocation of the action apply is not tail-recursive

Program	Generated compiler	GCC	Factor
fibo	2.81 s	1.50 s	1.9
ackerman	3.60 s	0.60 s	6.0
decrement	19.15 s	1.26 s	15.2
fibo-while	4.24 s	1.70 s	2.5
euclid	1.88 s	1.12 s	1.7
sieve	2.51 s	1.25 s	2.0
bubble	1.69 s	0.82 s	2.1

Table 10.2: miniC running times

will become less efficient when we extend the subset of C, especially if we allow more data than just integers and arrays of integers. Adding more features often means that the simple semantics of a construct is replaced by a more complex semantics, for instance, adding pointers and floats to the subset of C would mean that the semantic of + becomes more complicated because the operator should now be overloaded. On the other hand our compiler generates code that performs bounds checking on arrays, which the GCC compiler does not, which makes the generated compilers less efficient.

The test results in this section reveals that the efficiency of the generated compiler largely depends on the optimality of the ASD it is based on. It is also clear that the action compiler should be improved with respect to the implementation of the recursively combinator and the representation of data.

10.4.1 Comparison with OASIS

Comparing our result with the results achieved by others it is clear that our generated compilers are more efficient than all AS-based systems, except OASIS, but probably less efficient than compilers generated by Eli.

We have not been able to use the Ørbæk's OASIS system ourself because it is based on outdated software, and hardware we do not have access to. Our comparison is therefore based on the numbers listed in Table 10.3 taken from Section 4.5 in [93]. The numbers show the running times for *HypoPL* (a subset of Pascal) programs compiled with a generated compiler (second column), the running times for a similar program compiled with GCC with full optimisation (third column), and how much slower the output from the generated compiler is (fourth column).

Program	Generated compiler	GCC	Factor
fibo-while	0.8 s	0.5 s	1.6
sieve	1.2 s	0.3 s	4.0
euclid	2.1 s	0.7 s	3.0
bubble	0.4 s	0.2 s	2.0

Table 10.3: OASIS running times

We have implemented equivalent programs in miniC and the running times are displayed in Table 10.2. It is difficult to compare our action compiler with OASIS for various reasons:

- We are not able to test OASIS.
- OASIS uses another AN-based on an restricted version of the original AN and extended with extra features.
- Older hardware was used when measuring the running times for OASIS, so we can only compare the factor that its output is slower than GCC's output.
- The HypoPL language is a different from miniC. For instance, HypoPL allows neither functions with more than one argument nor mutually recursive functions.
- The results reported on OASIS are only with one decimal precision, so the factor might vary up to +/- 1 on some results.

All in all it is hardly fair to compare Ørbæk's results with ours. With this caveat we are going to try anyway. When comparing the numbers in the factor column in the last four rows of Tables 10.2 and the numbers in the factor row in Table 10.3, we notice that the output from our generated compiler is 2.1 times slower than gcc on the average, whereas OASIS's is 2.7 times slower. It would have been interesting to see how OASIS's generated compiler performs on the **ackerman** and **decrement** programs where we have significantly worser results.

10.5 Limitations

Both the use of actions as input and SML as output in our compiler generator puts some limitations on the languages that can be described. AS cannot describe all language features, for instance, `call/cc` known from many functional languages cannot be described in a straight forward way. The action compiler only accepts statically typed actions because it must be able to infer a type for the action. Our type inference algorithm puts further limitations on the set of actions accepted, for instance, actions originating from an ML program exploiting ML's let-polymorphism are not accepted. Finally the target language of the action compiler also limits the language features that can be described. The strict type system in SML means that it is difficult, if not impossible, to describe languages with subtypes.

Support for user defined data types is work in progress. Currently we support the data defined in the ASDF modules as part of a language description. It is only possible to describe languages where the user can define his own data types to some extent. The **length** program in Section 10.4 is an example of how the user can define a list data type in the Core ML language, but the description of data types in Core ML is not fully supported by the action compiler

yet, and only works on some examples. The representation of data is the main reason for performance loss in the generated compilers.

10.6 Conclusion and future work

We have presented an action compiler that, compared to previous results, is a small improvement with respect to the efficiency of the generated code. Our main contribution is the simplicity of the code generation where we use SML as target language.

Future work includes investigating how to generate code that is easier for the SML compiler to optimise. Especially the way data is represented in the generated code needs improvement. Relaxing the restrictions put on actions would also improve the system. Improving the type inference algorithm such that it accepts a bigger set of actions, would allow more natural descriptions of languages, but here we are also limited by the target language (SML) being strongly typed.

It would be interesting to see our compiler tested on the full Standard ML language or another realistic language, instead of just a sublanguage. Previous work has also only been tested on small languages; so far it has not been investigated how well action compilers scale to handle realistic programming languages. We think that at the moment van der Storm's compiler is the compiler that has the best chance of handling actions originating from a realistic language description because it can handle data that can be described using ATerms [10].

Using another target language is also worth investigating. There are compilers for Scheme and OCaml that on some examples produce faster code than MLton does on similar SML programs.

Chapter 11

Conclusion

One never notices what has been done;
one can only see what remains to be done.
— Marie Curie

We have presented a new formalism, ASDF, that supports writing constructive action semantic descriptions. Used in combination with the ASF+SDF formalism, we can write extensible, reusable descriptions of realistic programming languages, as we demonstrated in the Core ML example. The Action Environment supports working with both formalisms, and lets the user test language descriptions using the semantic function type checker and the action interpreter. Furthermore the Action Environment can be used to generate a front end for a compiler. The front end parses a language and maps its concrete syntax into actions. By combining the front end with the action compiler we have developed, we get a compiler for the described language; the Action Environment combined with an action compiler is an AS-based compiler generator.

All tools have been tested on the Core ML example, and the action compiler also on a subset of the C programming language named miniC.

The tests showed that the action compiler is an improvement with respect to efficiency compared to previous work on action compilers. The set of actions accepted by our compiler is acceptable compared to previous work; there exist some action compilers that accept more actions and others that accept less.

11.1 Future work

In this section we summarise the future work directions pointed out in previous chapters. For all tools it holds that they should be extended to also handle the actions that describe interactive processes (see Section 2.9).

The ASDF formalism is still at the experimental level and has only been tested on the Core ML example. Parameterised modules as found in SDF might result in greater reusability. Extending ASDF with support for user defined action abbreviations can help simplify the semantic equations. This would require extending the syntax used in the **requires** section so that the user can define the syntax of a new action combinator. Also the syntax of the semantic

equations should be extended so that the expansion of the abbreviation can be defined.

Allowing conditional equations, as found in ASF, in the **semantics** section would extend the expressiveness of ASDF. It is not clear whether it is needed since the construct described in an ASDF module should map easily to an action, and if conditional equations are needed, it might indicate that the construct is too complex and should be split into more constructs to improve the reusability of the constructs.

Using the ASF+SDF Meta-Environment as inspiration we can improve the functionality in the Action Environment. Features for refactoring ASDF modules should be implemented; this include renaming modules, adding and removing imports, and deleting modules. Validity check of ASDF modules can also be improved: besides the syntax check there should also be checks of whether the right-hand sides of equations use variables not occurring in the left-hand side. A language description might import two modules that define the same construct, but with different semantics. Functionality for merging the semantic equations of two such modules would improve the environment.

Type checkers can almost always be improved to accept a bigger set of legal programs; this also holds for our semantic function type checker. With respect to the user-friendliness of the type checker, it is worth considering whether our type system can become more transparent [24, page 7]; can the user predict whether a semantic function will type check, and is it clear what is wrong when the semantic function does not type check.

User-friendliness is also the main thing we need to improve in the action interpreter. Practical experience has shown that actions often unexpectedly evaluate to '*abrupt* ()' because there is an error in either the language specification or the program/action being evaluated, but this result does not say much about where in the program/action the problem lies. Instead of the trial and error approach, where we gradually simplify a program to find the bug, it would be helpful with an action debugger that allows breakpoints and stepwise evaluation of actions. Less important improvements of the interpreter include efficiency and a correct implementation of interleaving (the and action combinator) and the recursively combinator.

The action compiler and especially the type inference algorithm can be improved in several ways, and there is a lot of work to be done if we want to able to compile all actions resulting from the Core ML example. The set of actions that can be typed should be increased. As already mentioned, there are problems with the abruptly terminating actions, polymorphism, and user defined data types (more about this in Section 11.2). At the moment the type inference algorithm does not produce useful error reports when an action cannot be typed; this is not satisfactory if the action compiler should be used in practice.

The code generator can also be improved. Some of the test examples in Chapter 10 revealed inefficient implementations of especially data representation and the recursively combinator. Some of the inefficiency was caused by the inefficient ASD that produced the action given to the action compiler, but ideally the action compiler should also efficiently compile these cases. Although

we have tried to avoid doing transformations and static analysis on the actions, it might be necessary if we want to improve the compilation; depending on the ML compiler to do all optimisations might not be good enough.

In general more testing of our tools would help point out weaknesses. So far we have only tested them with the Core ML and the miniC examples. Testing a description of an object oriented language would be interesting, and would also show how many of the BAS constructs presented in Chapter 5 can be reused.

11.2 The success of AS-based compiler generation

AS-based compiler generation has been a research area for almost 15 years. Despite the many obvious advantages: easy to use formal specification language, tools for automatically generating a full compiler, guaranteed correctness of the generated compiler, etc., it has not gained much popularity outside academia. Most of the systems developed seem to have a very short lifetime, probably due to the lack of users and the developers moving to other areas of research. One exception is the Abaco system [72] which is still being developed by the AS group at Recife[1].

We think that there is several requirements that should be satisfied for AS-based compiler generation to become successful:

User-friendliness: For a compiler generator to become useful in practice, it is important that the time spent in learning to use the formalism and the tool is low enough to make it profitable to use the tool instead of implementing the compiler by hand. Furthermore the tools should make it easy to write language descriptions.

Error reporting: It is important that different tools that aid in the process of developing a language description give useful error reports.

Expressiveness: A compiler generator should use a formalism that can be used to describe a large set of languages, and the compiler generator should not put any limitations on the formalism.

Data description: Related to expressiveness is the ability to describe not just the value declarations, statements, and expressions, but also the data and types in a language.

Interfacing: Most languages contain parts where it is desirable to have more control over the implementation than an abstract description formalism allows. An interface to a general purpose language is needed.

Efficiency: It is important that the compiler generator, the generated compilers, and the code produced by the generated compilers are efficient. Efficiency can be measured both in execution time and code size.

[1] `http://www.cin.ufpe.br/~rat/`

An environment that supports developing a language description and an extensive library of reusable modules that describes common language features is our attempt to make a user-friendly language description system. The Action Environment lets the user define the concrete syntax, a mapping from concrete syntax to abstract syntax, and the AS of the abstract syntax. Instead of writing new action semantic descriptions of language constructs it is often possible to find a module in the library of BAS constructs (see Appendix C) that describes the construct and then import it in the language description by reference.

The number of formalisms used in a language description, the size of the formalisms, and the interfacing between them is also important when measuring the user-friendliness. In our system a language description employs two formalisms: ASF+SDF and ASDF. We think that the AS part of ASDF is the most time consuming part to learn. It is difficult to prove that a framework is user-friendly, but compared to frameworks using a more mathematical notation, we think that AS has an advantage in its use of English keywords. The only interfacing problems that should be handled by the user is the mapping from concrete syntax (defined using SDF) to abstract syntax (defined using ASDF), written in ASF, and this is in most cases a trivial mapping.

Part of a tool's user-friendliness is the quality of the error reports it delivers. Error reports can range from the error message saying that something went wrong, for instance, during a type check, to a precise description of where the problem is and perhaps a possible solution to the problem. Error reporting is an area where our tools can still be improved. The Action Environment can report parse errors by showing where in a file the error has occurred. The same holds for the semantic function type checker, but here the location is not always as precise. The action interpreter just informs the user that the action terminated abruptly if something went wrong during evaluation. The action compiler can only report that an action could not be compiled, but it does not explain why.

The set of languages and language features that can be handled by our compiler generator is both limited by the use of AS and the set of actions accepted by our action compiler. ASDs of many languages exists [21, 35, 41, 56, 69, 74, 90], but we are aware of two features which cannot easily be described: `call/cc`, as known from functional programming languages, and backtracking, as known from logic programming languages (in [50] it was suggested how to extend AN to support backtracking). The limitations of our action compiler have already been described in Chapter 10, for instance, it does not support all of the Core ML language. To our knowledge, none of the other action compilers have been tested on full descriptions of real world programming languages.

An important part of a programming language is support for user defined data types in the form of structures, classes, templates, etc. AS can describe declarations of user defined data types (see description of SML data types in [90] and Java classes in [21]), but the systems that performs type inference [23, 46, 75, 93] do not support it. Our action interpreter supports the user defined data types in Core ML, and it is likely that van der Storm's compiler [83] also supports user defined data types (it has not been tested), but achieving an efficient compilation of actions describing user defined data types is a challenge. The problem lies both in our type inference algorithm and the code generation.

The user might define recursive data types, subtype relations, intersection types, union types, or other features not supported by the type inference algorithm. The representation of user defined data types in our code generator is the main reason for inefficiency in our action compiler.

When developing a compiler, there will often be parts of the language which we want to optimise by implementing them by hand. Or there could be features that we cannot describe the semantics of in AS. Therefore it is important that the compiler generator supports an interface to an efficient general purpose language. The Abaco system [72] supports interfacing with Java, and van der Storm's AN-2 tools can be interfaced with both Java and C. Our system does not support any interfacing, but interfacing using SML's signatures should be investigated.

The last requirement for successful AS-based compiler generation is efficiency of the compiler generator. Efficiency with respect to running time can be measured in three ways: When describing a language, the time used to generate a compiler is important, but here an action interpreter is perhaps more appropriate for testing the language description. When writing a program, the time it takes for a generated compiler to compile the program is important, but again the action interpreter could also be used. Finally when using a program compiled with a generated compiler, the running time of the output from a generated compiler is important. In our work we have focused on the running time of the output of the generated compiler. The efficiency requirement probably collides with the expressiveness requirement and the data description requirement.

In our opinion, neither our own work nor previous work on AS-based compiler generation have been able to fully satisfy all of the above mentioned requirements, but it is our hope that our work has pushed the development towards an optimal AS-based compiler generator.

Bibliography

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. Iv, D. P. Friedman, E. Kohlbecker, J. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In Williams [92], pages 31–41.

[3] M. Anlauff, P. W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Proceedings of the 3rd International Conference on Perspectives of System Informatics, PSI'99*, LNCS volume 1755, pages 40–53. Springer-Verlag, 2000.

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

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

[6] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In Williams [92], pages 308–317.

[7] A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[8] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[9] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[10] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[11] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[12] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. Research Series BRICS RS-04-36, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2004. Extended version of [13], submitted to a special issue of *Science of Computer Programming* for LDTA'04 papers.

[13] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. In G. Hedin and E. V. Wyk, editors, *Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications, LDTA'04*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[14] M. G. J. van den Brand and P. Klint. *ASF+SDF Meta-Environment user manual, revision 1.149*, 2005. `http://www.cwi.nl/projects/MetaEnv/meta/doc/manual.ps.gz`.

[15] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, Amsterdam, 2001. `ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R0121.pdf`.

[16] M. G. J. van den Brand, S. Klusener, L. Moonen, and J. J. Vinju. Generalized parsing and term rewriting - semantics directed disambiguation. In B. Bryant and J. Saraiva, editors, *Proceedings of the 3rd Workshop on Language Descriptions Tools and Applications*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[17] M. G. J. van den Brand, P.-E. Moreau, and C. Ringeissen. The ELAN Environment: a rewriting logic environment based on ASF+SDF technology - system demonstration. In M. G. J. van den Brand and R. Lämmel, editors, *Proceedings of the 2nd Workshop on Language Descriptions, Tools and Applications, LDTA'02*, volume 65.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[18] M. G. J. van den Brand, P. E. Moreau, and J. J. Vinju. Environments for term rewriting engines for free! In R. Nieuwenhuis, editor, *Proceedings of 14th International Conference on Rewriting Techniques and Applications, RTA 2003*, LNCS volume 2706, pages 424–435. Springer-Verlag, 2003.

[19] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. N. Horspool, editor, *Compiler Construction*, LNCS volume 2304, pages 143–158. Springer-Verlag, 2002. `http://www.cs.ruu.nl/people/visser/ftp/BSVV02.pdf`.

[20] 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, *Proceedings of the 10th International Conference on Compiler Construction, CC 2001*, LNCS volume 2027, pages 365–370. Springer-Verlag, 2001.

[21] D. Brown and D. A. Watt. JAS: A Java Action Semantics. In Mosses and Watt [70], pages 43–55.

[22] D. F. Brown. *Sort inference in action semantics*. PhD thesis, Department of Computing Science, University of Glasgow, 1996.

[23] D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Conference on Compiler Construction, CC'92*, LNCS volume 641, pages 95–109. Springer-Verlag, 1992.

[24] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[25] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Research Report 94, INRIA, 1998.

[26] A. van Deursen. *Executable Language Definitions*. PhD thesis, CWI, Amsterdam, Holland, 1994. `http://www.cwi.nl/~arie/papers/pschrift.ps.gz`.

[27] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing volume 5. World Scientific, 1996.

[28] K.-G. Doh. Action transformation by partial evaluation. In W. L. Scherlis, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation, PEPM'95*, pages 230–240. ACM Press, 1995.

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

[30] K.-G. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In Krieg-Brückner [44], pages 151–166.

[31] K.-G. Doh and D. A. Schmidt. The facets of action semantics: Some principles and applications (extended abstract). In Mosses [57], pages 1–15.

[32] S. Even and D. A. Schmidt. Type inference for action semantics. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming, ESOP'90*, LNCS volume 432, pages 118–133. Springer-Verlag, 1990.

[33] P. Fritzson, editor. *Proceedings of the 5th International Conference on Compiler Construction, CC'94*, LNCS volume 786. Springer-Verlag, 1994.

[34] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[35] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In Mosses [57], pages 34–42.

[36] P. H. Hartel. LETOS – a lightweight execution tool for operational semantics. *Software, Practice & Experience*, 29(15):1379–1416, 1999.

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

[38] J. Iversen. Type inference for the new action notation. In Mosses [65], pages 78–98.

[39] J. Iversen. Type checking semantic functions in ASDF. Research Series BRICS RS-04-35, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2004. `http://www.brics.dk/RS/04/35/`.

[40] J. Iversen. An action compiler targeting Standard ML. In J. Boyland and G. Hedin, editors, *Proceedings of the 5th Workshop on Language Descriptions, Tools and Applications, LDTA'05*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005. to appear. Also available at `http://www.daimi.au.dk/~jive/papers/ldta2005.pdf`.

[41] J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *IEE Proceedings-Software special issue on Language Definitions and Tool Generation*, 2005. to appear.

[42] S. C. Johnson. YACC: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[43] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[44] B. Krieg-Brückner, editor. *Proceedings of the 4th European Symposium on Programming, ESOP'92*, LNCS volume 582. Springer-Verlag, 1992.

[45] S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In Mosses and Moura [67], pages 19–36. `http://www.brics.dk/~pdm/papers/LassenMossesWatt-AS-2000/`.

[46] K. D. Lee. *Action Semantics-based Compiler Generation*. PhD thesis, Department of Computer Science, University of Iowa, 1999. `http://www.cs.luther.edu/~leekent/papers/thesis.ps`.

[47] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In H. R. Nielson, editor, *Proceedings of the 6th European Symposium on Programming, ESOP '96*, LNCS volume 1058, pages 219–234. Springer-Verlag, 1996.

[48] A. Macedo and H. Moura. Investigating compiler generation systems. In *Proceedings of the 4th Brazilian Symposium on Programming Languages, SBLP 2000*, pages 259–266. Brazilian Computing Society, 2000.

[49] L. C. Menezes, H. P. de Moura, W. Cansanção, F. Lima, and L. Ribeiro. An action semantics integrated development environment. In M. G. J. van den Brand and D. Parigot, editors, *Proceedings of the 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01*, volume 44.2. Elsevier, 2001.

[50] L. C. Menezes, H. P. de Moura, and G. L. Ramalho. Action semantics for logic programming languages. In Mosses and Moura [67], pages 47–61.

[51] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[52] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.

[53] P. D. Mosses. Making denotational semantics less concrete. In *Proceedings of the International Workshop on Semantics of Programming Languages*, number Bericht nr. 41 in Abteilung Informatik, pages 102–109. Universität Dortmund, 1977.

[54] P. D. Mosses. SIS, Semantics Implementation System: Reference manual and user guide. Tech. Mono. MD–30, Dept. of Computer Science, Univ. of Aarhus, 1979.

[55] P. D. Mosses. Unified algebras and action semantics. In B. Monien and R. Cori, editors, *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science, STACS'89*, LNCS volume 349, pages 17–35. Springer-Verlag, 1989.

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

[57] P. D. Mosses, editor. *Proceedings of the 1st International Workshop on Action Semantics, AS'94*, BRICS NS-94-1. Dept. of Computer Science, Univ. of Aarhus, 1994.

[58] P. D. Mosses. Theory and practice of action semantics. In W. Penczek and A. Szalas, editors, *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science, MFCS '96*, LNCS volume 1113, pages 37–61. Springer-Verlag, 1996.

[59] P. D. Mosses. A tutorial on action semantics. 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).

[60] P. D. Mosses. Foundations of modular SOS. Research Series BRICS RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. `http://www.brics.dk/RS/99/54`. Full version of [61].

[61] P. D. Mosses. Foundations of Modular SOS (extended abstract). In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science, MFCS'99*, LNCS volume 1672, pages 70–80. Springer-Verlag, 1999.

[62] P. D. Mosses. AN-2: Revised action notation – syntax and semantics, 2000. Available at `http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/`.

[63] P. D. Mosses. Modularity in meta-languages. In J. Despeyroux, editor, *Proceedings of the 2nd Workshop on Logical Frameworks and Meta-Languages, LFM'00*, pages 1–18. INRIA, 2000.

[64] P. D. Mosses. The varieties of programming language semantics (and their uses). In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of the 4th International Conference on Perspectives of System Informatics, PSI 2001, Andrei Ershov Memorial Conference, Revised Papers*, LNCS volume 2244, pages 165–190. Springer-Verlag, 2001.

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

[66] P. D. Mosses. Definitive semantics. Available at `http://www.mimuw.edu.pl/~mosses/DS-03/Notes.pdf`, 2003.

[67] P. D. Mosses and H. 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.

[68] P. D. Mosses and D. 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.

[69] P. D. Mosses and D. A. Watt. Pascal action semantics, version 0.6. `ftp://ftp.brics.dk/Projects/AS/Papers/MossesWatt93DRAFT/pas-0.6.ps.Z`, 1993.

[70] P. D. Mosses and D. A. Watt, editors. *Proceedings of the 2nd International Workshop on Action Semantics, AS'99*, BRICS NS-99-3. Dept. of Computer Science, Univ. of Aarhus, 1999.

[71] H. Moura. *Action Notation Transformations*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, 1993.

[72] H. Moura, L. C. Menezes, M. Monteiro, P. Sampaio, and W. Cansanção. The Abaco system: An action tool for programming language designers. In Mosses [65], pages 1–8.

[73] H. Moura and D. A. Watt. Action transformations in the Actress compiler generator. In Fritzson [33], pages 16–30.

[74] M. A. Musicante. The Sun RPC language semantics. In *Proceedings of the 18th Latin-American Conference on Informatics, PANEL'92*, 1992.

[75] J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1992.

[76] J. Palsberg. A provably correct compiler generator. In Krieg-Brückner [44], pages 418–434.

[77] J. Palsberg and T. Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189(1):54–86, 2004.

[78] T. J. Parr and R. W. Quong. ANTLR: A predicated LL(k) parser generator. *Software, Practice & Experience*, 25(7):789–810, 1995.

[79] M. Pettersson. A compiler for natural semantics. In T. Gyimothy, editor, *Proceedings of the 6th International Conference on Compiler Construction, CC'96*, LNCS volume 1060, pages 177–191. Springer-Verlag, 1996.

[80] F. Pottier. A 3-part type inference engine. In G. Smolka, editor, *Proceedings of the 9th European Symposium on Programming Languages and Systems, ESOP'00*, LNCS volume 1782, pages 320–335. Springer-Verlag, 2000.

[81] S. Sankar, S. Viswanadha, and R. Duncan. JavaCC: The java compiler compiler. `https://javacc.dev.java.net/`.

[82] F. W. Schröer. *The Gentle Compiler Construction System*. R. Oldenbourg Verlag, Munich and Vienna, 1997. `http://gentle.compilertools.net/BOOK.ps.gz`.

[83] T. van der Storm. AN-2 tools. In Mosses [65], pages 23–42.

[84] M. Sulzmann. Designing record systems. Research Report YALEU/DCS/RR-1128, Yale University, Department of Computer Science, 1997.

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

[86] M. Tofte. *Compiler Generators: what they can do, what they might do, and what they will probably never do*, volume 19 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[87] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997. `http://www.cs.uu.nl/people/visser/ftp/Vis97.ps.gz`.

[88] J. Vollmer. Experiences with Gentle: Efficient compiler construction based on logic programming. In J. Maluszynski and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming – PLILP 1991*, LNCS volume 528, pages 425–426. Springer-Verlag, 1991.

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

[90] D. A. Watt. The static and dynamic semantics of SML. In Mosses and Watt [70], pages 155–172.

[91] J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, ICALP'02*, LNCS volume 2380, pages 913–925. Springer-Verlag, 2002. `http://www.macs.hw.ac.uk/~jbw/papers/Wells:The-Essence-of-Principal-Typings:ICALP-2002.pdf`.

[92] J. Williams, editor. *Proceedings of the Sixth Conference on Functional Programming Languages and Computer Architecture, FPCA'93*. ACM Press, 1993.

[93] P. Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Aarhus University, Computer Science Department, 1993. `ftp://ftp.daimi.au.dk/pub/empl/poe/oasis.ps.gz`.

[94] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In Fritzson [33], pages 1–15.

[95] P. Ørbæk. *Trust and Dependence Analysis*. PhD thesis, BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1997.

Appendix A

The syntax of Core ML

For readability we have used longer names in this appendix for the syntactic sorts introduced in Sect. 5.1. 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*.

A.1 Expressions

module *Expressions*

imports

Constants
Identifiers
Patterns
Declarations

exports

context-free start-symbols

Expression

sorts

Expression AtomicExp 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*

%% Expressions

ApplicationExp → *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

{*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*

context-free priorities

{ *Pattern ValueId Pattern*	→	*Pattern*} >
{"op"? *ValueId* (":" *Type*)? "as" *Pattern*	→	*Pattern*} >
{*Pattern* ":" *Type*	→	*Pattern*}

A.3 Declarations

module *Declarations*

imports

Expressions
Patterns

exports

context-free start-symbols

Declaration

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* }

Appendix B

Mapping ML to BAS

B.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 the result is then applied to a tuple of expressions. 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** *EX* := *ES*

[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*))

B.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* → *Par*
getlabels: { *SinglePatternRow* "," }+ → *Exp**
getpatts : { *SinglePatternRow* "," }+ → *Par**
pat2bas : { *Pattern* "," }+ → *Par**
pat2bas : *AtomicPattern+* → *Par+*
patcast : *Pattern* → *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 PA := PC

[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 AP1 := APS

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

B.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 '`v`*i*', where *i* is a positive integer. For more information about the mapping of function declarations consult Section 5.4.2

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, TypeConstructor)* → *Dec*
deccast	:	*SingleDataBinding* → *Dec*
dec2bas	:	*(SingleConsBinding, TypeConstructor)* → *Dec*
deccast	:	*(SingleConsBinding, TypeConstructor)* → *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] *dec2bas*(val *TVS VB*) = *dec2bas*(*VB*)

[val-2] *dec2bas*(*P* = *EX*) = *bind-val*(*pat2bas*(*P*), *exp2bas*(*EX*))

[val-3] *dec2bas*(*SVB* and *VB*) = *simult-seq*(*dec2bas*(*SVB*) *AD**)
when *simult-seq*(*AD**) := *dec2bas*(*VB*)

[val-4] *dec2bas*(rec *VB*) = *rec*(*dec2bas*(*VB*))

[default-val] *dec2bas*(*SVB* and *VB*) = *simult-seq*(*dec2bas*(*SVB*) *dec2bas*(*VB*))

[fun-1] *SFVBB* = *FVB*,
bind-val(*AP*, *AE1*) = *dec2bas*(*FVB*),
N = *numofargs*(*SFVBB*),
AE2 = *app-seq*(*abs*(*val*(valueid("f")), *curry*(*tuple-seq*(), *N*)), *AE1*)
dec2bas(fun *TVS FVB*) = *rec*(*bind-val*(*AP*, *AE2*))

[default-fun-1] *dec2bas*(fun *TVS FVB*) = *rec*(*dec2bas*(*FVB*))

[fun-2] *AE1* = *abs*(*tuple*(*pat2bas*(*ATP+*)), *exp2bas*(*EX*))
dec2bas(*op? I ATP+ CTY?* = *EX*) = *bind-val*(*var*(*I*), *AE1*)

[fun-3] *dec2bas*(*SFVB*) = *bind-val*(*AP1*, *AE1*),
dec2bas(*SFVBB*) = *bind-val*(*AP1*, *AE2*)
dec2bas(*SFVB* | *SFVBB*) = *bind-val*(*AP1*, *alt-seq*(*AE1 AE2*))

[fun-5] *bind-val*(*AP*, *AE1*) = *dec2bas*(*SFVBB*),
N = *numofargs*(*SFVBB*),
AE2 = *app-seq*(*abs*(*val*(valueid("f")), *curry*(*tuple-seq*(), *N*)), *AE1*)
simult-seq(*AD**) = *dec2bas*(*FVB*)
dec2bas(*SFVBB* and *FVB*) = *simult-seq*(*bind-val*(*AP*, *AE2*) *AD**)

[default-fun-5] *bind-val*(*AP1*, *AE1*) = *dec2bas*(*SFVBB*),
N1 = *numofargs*(*SFVBB*),
AE2 = *app-seq*(*abs*(*val*(valueid("f")),
curry(*tuple-seq*(), *N1*)), *AE1*)
bind-val(*AP2*, *AE3*) = *dec2bas*(*FVB*),
SFVBB2 = *FVB*,
N2 = *numofargs*(*SFVBB2*),
AE4 = *app-seq*(*abs*(*val*(valueid("f")),
curry(*tuple-seq*(), *N2*)), *AE3*)
dec2bas(*SFVBB* and *FVB*) =
simult-seq(*bind-val*(*AP1*, *AE2*) *bind-val*(*AP2*, *AE4*))

[fun-6] *dec2bas*(*SFVBB*) = *deccast*(*SFVB*) **when** *SFVB* := *SFVBB*

[fun-7] *dec2bas*(*FVB*) = *deccast*(*SFVBB*) **when** *SFVBB* := *FVB*

[numofargs-1] *numofargs*(*op? I ATP+ CTY?* = *EX* | *SFVBB*) = *length*(*ATP+*)

[numofargs-2] *numofargs*(*op? I ATP+ CTY?* = *EX*) = *length*(*ATP+*)

[length-1] *length*(*ATP1 ATP+*) = 1 + *length*(*ATP+*)

[length-2] *length*(*ATP1*) = 1

[default-curry-1] $\dfrac{N = \text{natcon}(CH+),\quad val(I) = val(\text{valueid}(\text{"v"}\,CH+))}{curry(\text{tuple-seq}(AE^*),\ N) = abs(var(I),\ curry(\text{tuple-seq}(AE^*\ val(I)),\ N\text{-}1))}$

[curry-2] *curry*(*AE*, 0) = *app-seq*(*val*(valueid("f")), *AE*)

[cast-1] *deccast*(*SFVB*) = *dec2bas*(*SFVB*)

[cast-2] *deccast*(*SFVBB*) = *dec2bas*(*SFVBB*)

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

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

[datatype-2] *dec2bas*(*TVS TC* = *CB*) = *dec2bas*(*CB*, *TC*)

[datatype-3] *dec2bas*(*SDB* and *DB*) = *simult-seq*(*dec2bas*(*SDB*) *AD**)
when *simult-seq*(*AD**) := *dec2bas*(*DB*)

[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*, *TC*) = *bind-val*(*pat2bas*(*I*), *new-cons*(*TC*))
when *op? I OTY?* := *SCB*

[cons-bind-2] *dec2bas*(*SCB* | *CB*, *TC*) = *simult-seq*(*dec2bas*(*SCB*, *TC*) *AD**)
when *simult-seq*(*AD**) := *dec2bas*(*CB*, *TC*)

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

[cons-bind-4] *dec2bas*(*CB*, *TC*) = *deccast*(*SCB*, *TC*) **when** *SCB* := *CB*

[cast-5] *deccast*(*SCB*, *TC*) = *dec2bas*(*SCB*, *TC*)

[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(exn)*)
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 *simult-seq*(*AD**) := *dec2bas*(*EB*)

[default-exception-5] *dec2bas*(*SEB* and *EB*) =
simult-seq(*dec2bas*(*SEB*) *dec2bas*(*EB*))

[exception-6] *dec2bas*(*EB*) = *deccast*(*SEB*) **when** *SEB* := *EB*

[cast-6] *deccast*(*SEB*) = *dec2bas*(*SEB*)

[local-1] *dec2bas*(local *D1* in *D2* end) = *local*(*dec2bas*(*D1*), *dec2bas*(*D2*))

[open-1] *dec2bas*(open *LS*) = *ignore*

[seq-1] *dec2bas*(*D1* ; *D2*) = *accum*(*dec2bas*(*D1*) *dec2bas*(*D2*))

[seq-2] *dec2bas*(*D1 D2*) = *accum*(*dec2bas*(*D1*) *dec2bas*(*D2*))

[infix-1] *dec2bas*(infix *d? I+*) = *ignore*

[infix-2] *dec2bas*(infixr *d? I+*) = *ignore*

[infix-2] *dec2bas*(nonfix *I+*) = *ignore*

Appendix C

Action semantic descriptions of BAS constructs

Module 1 *CoreML*

imports

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
Exp/New-Cons

Stm/Exp
Stm/While

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

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

C.1 Expressions

Module 2 *Exp*

requires

E : *Exp*

semantics evaluate: *Exp* → *Action* & using data & giving val

Module 3 *Exp/Val*

syntax *Exp* **::=** *Val*

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

Module 4 *Exp/Val-Id-Const*

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

requires

Val **::=** *Cons*

Bindable **::=** *Val*

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

Module 5 *Exp/App-Seq*

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

requires

Val **::=** *Func* | *ConsData* | *Cons*

func-no-apply : *Val*

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

Module 6 *Exp/Tuple-Seq*

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

requires *Val* **::=** *Tuple*

empty-tuple : *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*

Module 14 *Exp/New-Cons*

syntax

Exp ::= new-cons(*Token*)

requires

Val ::= *Func*

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

semantics

evaluate new-cons(T) = create 0 then give (cons(datacons(the token T, the cell)))

C.2 Statements

Module 15 *Stm*

requires S : *Stm*

semantics execute : *Stm* → *Action* & using data & giving ()

Module 16 *Stm/Exp*

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

semantics execute stm(E) = evaluate E then skip

Module 17 *Stm/While*

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

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

semantics execute while(E, S) =
unfolding (evaluate E then
maybe check (not(the boolean))
then skip
else (execute S then unfold))

C.3 Parameters

Module 18 *Par*

requires *P* : *Par*

semantics match : *Par* → *Action* & using val & giving bindings

Module 19 *Par/Val*

syntax *Par* **::=** *Val*

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

Module 20 *Par/Val-Or-Var*

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

requires

Val **::=** *Cons*

Bindable **::=** *Val*

semantics match val-or-var(*T*) =
(maybe check (val(the cons bound-to the token *T*) = the val)
then give no-bindings)
else
(give (the cons bound-to the token *T*) then fail
and-catch bind(the token *T*, the val))

Module 21 *Par/Var*

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

requires *Bindable* **::=** *Val*

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

Module 22 *Par/Anon*

syntax *Par* **::=** anon

semantics match anon = give no-bindings

Module 23 *Par/App*

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

requires

Val **::=** *Func* | *ConsData* | *Cons*

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 24 *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 25 *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

C.4 Declarations

Module 26 *Dec*

requires *D* : *Dec*

semantics declare : *Dec* → *Action* & using data & giving bindings

Module 27 *Dec/Bind-Val*

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

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

Module 28 *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 29 *Dec/Rec*

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

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

Module 30 *Dec/Local*

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

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

Module 31 *Dec/Accum*

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

semantics

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

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

Module 32 *Dec/Ignore*

syntax *Dec* **::=** ignore

semantics declare ignore = give no-bindings

C.5 Data

Module 33 *Data/Bindings*

Module 34 *Data/Boolean*

Module 35 *Data/Cons*

requires

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

Module 36 *Data/Func*

requires

Func **::=** func(action: *Action* & using val & giving val)
| datacons(token: *Token*, tag: *Cell*)

Module 37 *Data/Tuple*

requires

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

C.6 Miscellaneous

Module 38 *Token*

requires

T : *Token*

Module 39 *Val*

requires

V : *Val*

Appendix D

ASF equations for evaluating actions

[give-infix-action] $$eval(\textsf{give } IA,\ (A1,\ A2),\ BS,\ E) = \\ <normal\ (A1\ IA\ A2),\ E> \tag{D.1}$$

[give-prefix-action] $$eval(\textsf{give } PA,\ A,\ BS,\ E) = <normal\ (PA\ A),\ E> \tag{D.2}$$

[check-true] $$\frac{normal\ true := eval\text{-}dataop(DO,\ D,\ E)}{eval(\textsf{check } DO,\ D,\ BS,\ E) = <normal\ D\ ,\ E>} \tag{D.3}$$

[check-false] $$\frac{normal\ false := eval\text{-}dataop(DO,\ D,\ E)}{eval(\textsf{check } DO,\ D,\ BS,\ E) = <abrupt\ ()\ ,\ E>} \tag{D.4}$$

[check-abrupt] $$\frac{abrupt\ D2 := eval\text{-}dataop(DO,\ D1,\ E)}{eval(\textsf{check } DO,\ D1,\ BS,\ E) = <abrupt\ D2,\ E>} \tag{D.5}$$

[default-and-right] $$\frac{\begin{array}{l}<normal\ D2,\ E2> := eval(A1,\ D1,\ BS,\ E1),\\ <R,\ E3> := eval(A2,\ D1,\ BS,\ E2),\\ normal\ D3\ \mathrel{!}:=\ R\end{array}}{eval(A1\ \textsf{and}\ A2,\ D1,\ BS,\ E1) = <R,\ E3>} \tag{D.6}$$

[default-and-left] $$\frac{\begin{array}{l}<R,\ E2> := eval(A1,\ D1,\ BS,\ E1),\\ normal\ D2\ \mathrel{!}:=\ R\end{array}}{eval(A1\ \textsf{and}\ A2,\ D1,\ BS,\ E1) = <R,\ E2>} \tag{D.7}$$

[and-then] $$eval(A1 \text{ and-then } A2,\ D,\ BS,\ E) = eval(A1 \text{ and } A2,\ D,\ BS,\ E) \tag{D.8}$$

[indivisibly] $$eval(\text{indivisibly } A,\ D,\ BS,\ E) = eval(A,\ D,\ BS,\ E) \tag{D.9}$$

[catch] $$\frac{<abrupt\ D2,\ E2> := eval(A1,\ D1,\ BS,\ E1) \quad <R,\ E3> := eval(A2,\ D2,\ BS,\ E2)}{eval(A1 \text{ catch } A2,\ D1,\ BS,\ E1) = <R,\ E3>} \tag{D.10}$$

[default-catch] $$\frac{<R,\ E2> := eval(A1,\ D1,\ BS,\ E1),\ abrupt\ D2\ !:=\ R}{eval(A1 \text{ catch } A2,\ D1,\ BS,\ E1) = <R,\ E2>} \tag{D.11}$$

[and-catch] $$\frac{<abrupt\ D2,\ E2> := eval(A1,\ D1,\ BS,\ E1),\ <abrupt\ D3,\ E3> := eval(A2,\ D1,\ BS,\ E2)}{eval(A1 \text{ and-catch } A2,\ D1,\ BS,\ E1) = <abrupt\ D2{+}D3,\ E3>} \tag{D.12}$$

[default-and-catch-right] $$\frac{<abrupt\ D2,\ E2> := eval(A1,\ D1,\ BS,\ E1),\ <R,\ E3> := eval(A2,\ D1,\ BS,\ E2),\ abrupt\ D3\ !:=\ R}{eval(A1 \text{ and-catch } A2,\ D1,\ BS,\ E1) = <R,\ E3>} \tag{D.13}$$

[default-and-catch-left] $$\frac{<R,\ E2> := eval(A1,\ D1,\ BS,\ E1),\ abrupt\ D2\ !:=\ R}{eval(A1 \text{ and-catch } A2,\ D1,\ BS,\ E1) = <R,\ E2>} \tag{D.14}$$

[else] $$\frac{<fail,\ E2> := eval(A1,\ D,\ BS,\ E1) \quad <R,\ E3> := eval(A2,\ D,\ BS,\ E2)}{eval(A1 \text{ else } A2,\ D,\ BS,\ E1) = <R,\ E3>} \tag{D.15}$$

$$\text{[default-else]}\quad \frac{<R,\ E2> := eval(A1,\ D,\ BS,\ E1) \quad fail\ !=\ R}{eval(A1\ \mathsf{else}\ A2,\ D,\ BS,\ E1) = <R,\ E2>} \tag{D.16}$$

$$\text{[choose-nat]}\quad \frac{<N,\ E2> := next\text{-}random(E1)}{eval(\mathsf{choose\text{-}nat},\ D,\ BS,\ E1) = <normal\ N,\ E2>} \tag{D.17}$$

$$\text{[scope]}\quad \frac{<normal\ BS2,\ E2> := eval(A1,\ D,\ BS1,\ E1) \quad <R,\ E3> := eval(A2,\ D,\ BS2,\ E2)}{eval(A1\ \mathsf{scope}\ A2,\ D,\ BS1,\ E1) = <R,\ E3>} \tag{D.18}$$

$$\text{[default-scope-1]}\quad \frac{<normal\ D2,\ E2> := eval(A1,\ D1,\ BS1,\ E1), \quad BS2\ !:=\ D1}{eval(A1\ \mathsf{scope}\ A2,\ D1,\ BS1,\ E1) = <abrupt\ (),\ E2>} \tag{D.19}$$

$$\text{[default-scope-2]}\quad \frac{<R,\ E2> := eval(A1,\ D1,\ BS,\ E1), \quad normal\ D2\ !:=\ R}{eval(A1\ \mathsf{scope}\ A2,\ D1,\ BS,\ E1) = <R,\ E2>} \tag{D.20}$$

$$\text{[create]}\quad \frac{normal\ D2 := eval(\mathsf{the\ storable},\ D1,\ E) \quad <C,\ E2> := create\text{-}cell(D1,\ E1)}{eval(\mathsf{create},\ D1,\ BS,\ E1) = <normal\ C,\ E2>} \tag{D.21}$$

$$\text{[default-create]}\quad eval(\mathsf{create},\ D,\ BS,\ E) = <abrupt\ (),\ E> \tag{D.22}$$

$$\text{[inspect]}\quad \frac{D\ !:=\ inspect\text{-}cell(C,\ E)}{eval(\mathsf{inspect},\ C,\ BS,\ E) = <normal\ D,\ E>} \tag{D.23}$$

$$\text{[default-inspect]}\quad eval(\mathsf{inspect},\ D,\ BS,\ E) = <abrupt\ (),\ E> \tag{D.24}$$

$$\text{[default-apply]} \quad \frac{(A,\ D^*)\ !:=\ D}{eval(\mathsf{apply},\ D,\ BS,\ E) = <abrupt\ (),\ E>} \tag{D.25}$$

$$\text{[default-close]} \quad \frac{A\ !:=\ D}{eval(\mathsf{close},\ D,\ BS,\ E) = <abrupt\ (),\ E>} \tag{D.26}$$

Appendix E

Code rules

$$\frac{O \longrightarrow E}{\mathsf{give}\ O \longrightarrow E} \tag{E.1}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ n_1 = |normout(\mathcal{T}(A_1))|,\ n_2 = |normout(\mathcal{T}(A_2))| \end{array}}{\begin{array}{l} A_1\ \mathsf{and}\ A_2 \longrightarrow\ \texttt{fn (t, b) =>} \\ \qquad \texttt{let val } (d_1, \ldots,\ d_{n_1}) \texttt{ = } E_1\texttt{(t, b);} \\ \qquad\quad \texttt{val } (d_{n_1+1}, \ldots,\ d_{n_1+n_2}) \texttt{ = } E_2\texttt{(t, b)} \\ \qquad \texttt{in } (d_1, \ldots,\ d_{n_1+n_2}) \texttt{ end} \end{array}} \tag{E.2}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ \emptyset = normout(\mathcal{T}(A_1)) \end{array}}{A_1\ \mathsf{and}\ A_2 \longrightarrow E_1} \tag{E.3}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ \emptyset \neq normout(\mathcal{T}(A_1)),\ \emptyset = normout(\mathcal{T}(A_2)) \end{array}}{\begin{array}{l} A_1\ \mathsf{and}\ A_2 \longrightarrow\ \texttt{fn (t, b) =>} \\ \qquad \texttt{let val \_ = } E_1\texttt{(t, b);} \\ \qquad\quad \texttt{val \_ = } E_2\texttt{(t, b)} \\ \qquad \texttt{in () end} \end{array}} \tag{E.4}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ \emptyset = normout(\mathcal{T}(A_1)) \end{array}}{A_1\ \mathsf{and\text{-}then}\ A_2 \longrightarrow E_1} \tag{E.5}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ \emptyset \neq normout(\mathcal{T}(A_1)),\ \emptyset = normout(\mathcal{T}(A_2))\end{array}}{\begin{array}{l}A_1 \text{ and-then } A_2 \longrightarrow \texttt{ fn (t, b) =>} \\ \qquad\qquad \texttt{let val \_ =} E_1\texttt{(t, b);} \\ \qquad\qquad\quad \texttt{val \_ =} E_2\texttt{(t, b)} \\ \qquad\qquad \texttt{in () end}\end{array}} \tag{E.6}$$

$$\frac{A \longrightarrow E}{\text{indivisbly } A \longrightarrow E} \tag{E.7}$$

$$\frac{I = excepid(abruptout(\mathcal{T}(\text{throw})))}{\text{throw} \longrightarrow \texttt{fn (t, b) => raise } I \texttt{ t}} \tag{E.8}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ \emptyset = abruptout(\mathcal{T}(A_1))\end{array}}{A_1 \text{ catch } A_2 \longrightarrow E_1} \tag{E.9}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I_1 = excepid(abruptout(\mathcal{T}(A_1))), I_2 = excepid(abruptout(\mathcal{T}(A_2))) \\ n_1 = |abruptout(\mathcal{T}(A_1))|,\ n_2 = |abruptout(\mathcal{T}(A_2))|\end{array}}{\begin{array}{l}A_1 \text{ and-catch } A_2 \longrightarrow \texttt{ fn (t, b) => (}E_1\texttt{ (t, b)} \\ \qquad\qquad \texttt{handle } I_1\texttt{ (}d_1, \ldots, d_{n_1}\texttt{) => (}E_2\texttt{ (t, b)} \\ \qquad\qquad\quad \texttt{handle } I_2\texttt{ (}d_{n_1+1}, \ldots, d_{n_1+n_2}\texttt{)} \\ \qquad\qquad\qquad \texttt{=> (}d_1, \ldots, d_{n_1+n_2}\texttt{)))}\end{array}} \tag{E.10}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ \emptyset = abruptout(\mathcal{T}(A_1))\end{array}}{A_1 \text{ and-catch } A_2 \longrightarrow E_1} \tag{E.11}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I_1 = excepid(abruptout(\mathcal{T}(A_1))), \emptyset = abruptout(\mathcal{T}(A_2))\end{array}}{\begin{array}{l}A_1 \text{ and-catch } A_2 \longrightarrow \texttt{ fn (t, b) => (}E_1\texttt{ (t, b)} \\ \qquad\qquad \texttt{handle } I_1\texttt{ \_ => (}E_2\texttt{ (t, b)))}\end{array}} \tag{E.12}$$

$$\text{fail} \longrightarrow \texttt{fn (t, b) => raise FAIL} \tag{E.13}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \end{array}}{A_1 \text{ else } A_2 \longrightarrow \texttt{fn (t, b) => (}E_1\texttt{ (t, b) handle FAIL => }E_2\texttt{ (t, b))}} \quad \text{(E.14)}$$

$$\text{choose-nat} \longrightarrow \texttt{fn (t, b) => random ()} \quad \text{(E.15)}$$

$$\text{inspect} \longrightarrow \texttt{fn (t, b) => !t} \quad \text{(E.16)}$$

$$\text{update} \longrightarrow \texttt{fn ((c, d), b) => c := d} \quad \text{(E.17)}$$

Recent BRICS Dissertation Series Publications

DS-05-2 **Jørgen Iversen.** ***Formalisms and tools supporting Constructive Action Semantics*****. May 2005. PhD thesis. xii+189 pp.**

DS-05-1 **Kirill Morozov.** ***On Cryptographic Primitives Based on Noisy Channels*****. March 2005. PhD thesis. xii+102 pp.**

DS-04-6 **Paweł Sobocinski.** ***Deriving Process Congruences from Reaction Rules*****. December 2004. PhD thesis. xii+216 pp.**

DS-04-5 **Bjarke Skjernaa.** ***Exact Algorithms for Variants of Satisfiability and Colouring Problems*****. November 2004. PhD thesis. x+112 pp.**

DS-04-4 **Jesper Makholm Byskov.** ***Exact Algorithms for Graph Colouring and Exact Satisfiability*****. November 2004. PhD thesis.**

DS-04-3 **Jens Groth.** ***Honest Verifier Zero-knowledge Arguments Applied*****. October 2004. PhD thesis. xii+119 pp.**

DS-04-2 **Alex Rune Berg.** ***Rigidity of Frameworks and Connectivity of Graphs*****. July 2004. PhD thesis. xii+173 pp.**

DS-04-1 **Bartosz Klin.** ***An Abstract Coalgebraic Approach to Process Equivalence for Well-Behaved Operational Semantics*****. May 2004. PhD thesis. x+152 pp.**

DS-03-14 **Daniele Varacca.** ***Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation*****. November 2003. PhD thesis. xii+163 pp.**

DS-03-13 **Mikkel Nygaard.** ***Domain Theory for Concurrency*****. November 2003. PhD thesis. xiii+161 pp.**

DS-03-12 **Paulo B. Oliva.** ***Proof Mining in Subsystems of Analysis*****. September 2003. PhD thesis. xii+198 pp.**

DS-03-11 **Maciej Koprowski.** ***Cryptographic Protocols Based on Root Extracting*****. August 2003. PhD thesis. xii+138 pp.**

DS-03-10 **Serge Fehr.** ***Secure Multi-Player Protocols: Fundamentals, Generality, and Efficiency*****. August 2003. PhD thesis. xii+125 pp.**

DS-03-9 **Mads J. Jurik.** ***Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols*****. August 2003. PhD thesis. xii+117 pp.**

DS-03-8 **Jesper Buus Nielsen.** ***On Protocol Security in the Cryptographic Model*****. August 2003. PhD thesis. xiv+341 pp.**