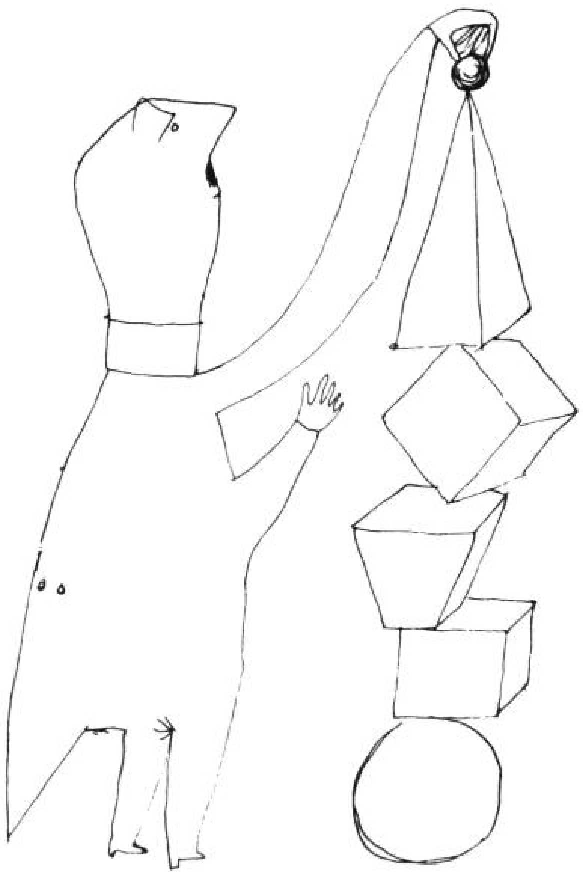
Implementing Actions

Tijs van der Storm



Cover image taken from: Franciszka Themerson, *The Way it Walks*, Gaberbocchus/Turret, London, 1988. Subscriptum:

> *When a person does something voluntarily, in the sense that he does it on purpose or is trying to do it, his action certainly reflects some quality or qualities of mind, since (it is more than a verbal point to say) he is in some degree minding what he is doing.*
> Gilbert Ryle, The Concept of Mind, p. 74.

Implementing Actions

Tijs van der Storm

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica (FNWI)
Afstudeerrichting: Programmatuur
Afstudeerdocent: Prof.dr. P. Klint

11. Think of the tools in a tool-box: there is a hammer, pliers, a saw, a screw-driver, a rule, a glue-pot, glue, nails and screws.—The functions of words are as diverse as the functions of these objects. (And in both cases there are similarities.)

Of course, what confuses us is the uniform appearance of words when we hear them spoken or meet them in script and print. For their *application* is not presented to us so clearly. [...]

291. What we call "*descriptions*" are instruments for particular uses. Think of a machine-drawing, a cross-section, an elevation with measurements, which an engineer has before him. Thinking of a description as a word-picture of the facts has something misleading about it: one tends to think only of such pictures as hang on our walls: which seem simply to portray how a thing looks, what it is like. (These pictures are as it were idle.)

Ludwig Wittgenstein, *Philosophical Investigations*

Foreword

When I arrived at Paul Klint's office to initiate a graduation project, I had no idea what the subject of my thesis should be. I asked Paul if he had any ideas. When he mentioned Action Semantics and described it as a user friendly formalism to define programming languages I became enthusiastic. Being interested in programming languages in general and ASF+SDF in specific, Action Semantics seemed perfect. It was decided to start the development of an interpreter for Action Notation. If this turned out to be easy, the implementation of a compiler would be the next step. Eventually the result comprised three interpreters and three compilers. My enthusiasm had become almost relentless. Friends of mine reminded me that I had called this thesis' subject "the best possible",—I must surely have been drunk. Nevertheless I enjoyed working on Action Semantics very much and this is partly due to the support I received from a number of people.

First of all I want to thank Paul Klint for proposing the subject of this thesis. He was always available to monitor the direction of my work, even when I kept failing his mild deadlines. Paul was also the one who now and then reminded me to pace down a little. Without him this thesis would never have arrived at the final stage. Finally it was through him that I received the invitation of Peter Mosses (the founding father of Action Semantics) to take part in a workshop on Action Semantics in Kopenhagen. Participating in the workshop was very stimulating and I thank both Peter and Paul for the opportunity and confidence.

Jurgen Vinju was always there when I had problems with or questions about the ASF+SDF Meta-Environment. This was both useful as well as inspiring. We also had discussions about science in the large (including the humanities) and software engineering in particular,—this put things in perspective again.

Finally I want to thank Susanna, my love, who has been a constant source of support. She never complained when I worked a night through and always showed genuine interest when I tried to explain Action Semantics. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To introduce this thesis, let's highlight some position statements taken from a paper *What Use Is Formal Semantics?* [35] by Peter Mosses. The paper can be seen as a practical assessment of the various semantical formalisms that exist. Mosses states that in contrast to formal syntax, formal semantics is "almost never used in practical applications". The fact is surprising since formal semantics is practised widely in the theoretical realm of computer science as well as in education. The reason seems to be the lack of user-friendliness of most semantic frameworks. Mosses concludes that user-friendliness of semantic frameworks encourages practical use of formal semantics. Among user-friendly features are: modularity, compositionality and readability. For example, neither denotational nor Structured Operational Semantics (SOS) is modular. Operational Semantics in general is not compositional; denotational semantics *is* compositional but is very unreadable (large λ-terms) and original computational concepts are hard to recover. Mosses states that Action Semantics might be a good candidate for practical use, since it combines all these features. Now the question emerges: "Well, is Action Semantics *really* usable in practical applications?"

That is the question this thesis tries to answer.

To answer this question we have to elaborate a little on the meaning of *use*, because the possible uses of semantics are manifold. One can think of prototyping a new language, documenting an existing language in a precise way or proving properties about the language using the semantics. In this thesis we are concerned with the generation of compilers and interpreters from Action Semantic descriptions. At the same time we concentrate on the *practical use* of Action Semantics as a formal framework for the definition of Domain Specific Languages (DSLs) [10, 11, 12, 13]. All the other uses of Action Semantics stand, of course, unaffected.

In a nutshell, the key questions are:

- Is the *new* Action Notation a good starting point to generate interpreters and compilers from language descriptions?
- What are the options of carrying out the task of generating a compiler or interpreter? Which strategies lead to the most practical solutions?
- How can the generated tools be embedded in existing software environments?

During the design and implementation of the tools we focussed on a number of aspects that we deemed important ‘side effects’ of our research question:

- Deployment: generated tools should be as deployable as possible. First this means that the tools should be platform independent. Secondly connection to other software (e.g. user interfaces, file systems, databases etc.) must be possible without much effort.
- Engineering: the compilers and interpreters should be relatively maintainable. Due to the possible changes in the Action Notation and Data Notation it should be easy to change or adapt the tools.
- Performance: last but not least, the tools derived from semantic descriptions should be reasonably efficient.

Organisation

Some notes about the organization of this thesis are in order. The next chapter provides an introduction to the new Action Notation (AN2) and how it can be used to define a small programming language. Chapter 3 is a revised version of a paper that has appeared in the proceedings of AS2002,—a workshop held in Kopenhagen in July 2002 as part of FLoC’02 [40]. The chapter surveys the tools that have been developed and presents some performance results. The chapters that follow can be seen as in depth discussions of the subjects covered in chapter 3. Chapter 4 dives into the technology of the two term rewriting approaches to executing actions. It also provides a mild introduction to the theory of Modular SOS. Then, in Chapter 5 the architecture of two compilers is described: `acc` for compiling actions to C, and `ajc` which compiles action to Java. Chapter 6 deals with the compilation of actions to an intermediate bytecode format and the involved machinery. Finally, the last chapter presents some conclusions.

Chapter 2

Introducing Action Semantics

2.1 Introduction

This chapter provides some background to the *practice* of giving an action semantics to a (simple) programming language, using ASF+SDF. At the same time the basic concepts of action semantics and the new Action Notation (AN2) are explained. The toy language PICO$^-$, a subset of the well known language Pico, is used as an illustrating vehicle. First we present a global overview of Action Semantics and the new Action Notation (AN2) and discuss some differences between the new AN2 and the old AN1. The second part of this chapter is dedicated to the action semantic description of PICO$^-$.

2.2 Background of Action Semantics

Action Semantics evolved from the need for a readable, yet formal, framework for describing programming language semantics that was able to scale up to the definition of real-life programming languages [29]. It particularly provided solutions to a number of problems encountered with other semantics formalisms. The most popular styles are operational semantics (small step and big step) and denotational semantics. Action Semantics promises to be the best of both worlds: a *compositional* semantics with a straightforward *operational* meaning. As such it can be characterized as a hybrid approach to semantics. Furthermore, Action Semantics particularly addressed pragmatic problems encountered with denotational semantics. First of all, (prototype) implementations of programming languages based on denotational semantics suffer from lack of performance. In denotational semantics an abstract syntax tree is transformed into one big lambda term the reduction of which is very costly. A related problem is that original concepts of computation (such as storage updates) are hard to discern in such lambda terms, which is an obstacle to analysis and optimization. Action Semantics solves this problem by using concise, englishlike denotations which capture precisely these concepts of computation: *actions.* This *Action Notation* thus enables better optimization opportunities when generating compilers and interpreters [9, 38, 8]. Another important pragmatic advantage of Action Notation is its modularity, which encourages reuse and separation of concerns.

For the first version of action notation (AN1) unified algebras were used to define abstract syntax and data operations [28]. Its semantics was defined using regular SOS. Unified algebras also provided a formalism to write so called Action Semantic Descriptions (ASDs) to define programming languages. In the new proposed version of action notation (AN2), unified algebras are left behind. The semantics of AN2 is defined in a new flavour of Structural Operational Semantics: Modular SOS. This specification is presented in Casl which also allows the definition of mixfix abstract syntax. The difference between MSOS and conventional SOS lies in the fact that a labelled transition relation is defined between (abstract) syntax and computed values only; auxiliary structures (stores, environments, etc.) are located on the *labels* of the transition relation. These labels form arrows of a category. Two transitions are allowed to be adjacent, when the two labels can be composed. The advantage of this way of dealing with additional constructs is a high degree of modularity. When a language changes, the MSOS specification can be modified accordingly without having to reformulate every transition rule. We will dive a bit deeper into the theory of MSOS when discussing the derivation of an interpreter in Asf+Sdf from the specification of AN2 in MSOS.

2.2.1 The new Proposal: AN2

Although AN2 has not yet emerged from the stadium of proposal, it is sufficiently rich to allow the description of most highlevel languages. Unlike the older version of Action Notation (AN1) AN2 can be vertically divided in two layers: Full AN2 and Kernel AN2 (AN-K). Full AN2 is defined by reduction to AN-K. The kernel of AN2 is a lot smaller than Full AN2 and is defined in Modular SOS. As a result the semantics of AN-K is a lot simpler and easier to deal with than that of AN1 [27]. The facets that divided AN1 in a horizontal way are still present in AN2, albeit slightly different.

1. Data and Control flow: functional computation, selection, exceptions.

2. Declarative: flow of bindings and scoping.

3. Reflective: actions as data, closure.

4. Imperative: effects on storage.

5. Interactive: processes and communication.

A difference with AN1 is that bindings are now included in the sort of individual data values (the sort *Datum*). That means that binding a token to a value is performed using *data operations*. Similarly, actions themselves are subsort of *Datum* without the use of an abstraction operator.

The central concept of Action Semantics is the *action*. Actions can be *atomic* or *combined* using prefix or infix *combinators*. If an action is *performed* it may *terminate*. Atomic actions always terminate in one execution step, while a combined action may take more steps if it terminates at all. If an action terminates, it either terminates *normally*, *exceptionally* or *failing*. Actions that terminate normally *give* data (*given* data). If an action terminates exceptionally,

	Data- and Controlflow
provide d	giving constant data
copy	copying given data
A_1 **then** A_2	functional composition
A_1 **and then** A_2	sequential composition
A_1 **and** A_2	interleaving
indivisibly A_2	anti-interleaving
raise	raising an exception
A_1 **exceptionally** A_2	exceptional composition
A_1 **and exceptionally** A_2	exceptional sequential composition
give o	computing dataoperations
check q	testing datapredicates
fail	abandoning an action
A_1 **otherwise** A_2	alternative composition
select (A_1 **or** ... **or** A_n)	nondeterministic choice
choose natural	arbitrary choice
	Scopes and Bindings
give current bindings	current bindings as data
A_1 **hence** A_2	scoping of bindings
	Reflection
enact	performance of a given action
	Storage
create	allocating and initializing a cell
destroy	deallocating a cell
update	updating a cell
inspect	inspecting a cell
	Interactive processes
activate	activating a new agent
deactivate	deactivating an agent
give current agent	current agent as data
send	sending a message to an agent
receive	receiving a message from an agent
give current time	current time as data

Table 2.1: Kernel AN2

the action is said to *raise* data (*raised* data). The termination mode of an action is called its *outcome*. Finally, all actions *take* given data as input. The complete kernel of AN is displayed in Table 2.1. For the full AN abbreviations we refer to the Appendix.

2.3 Action Semantics for Pico$^-$

2.3.1 Structure of the Specification of Pico$^-$

Pico$^-$ is a small language similar to While. It features integer expressions, static (single) scoping, assignment, while statements, and if statements. The only difference to 'full' Pico is the absence of string expressions. String expressions have been excluded because of a number of small pragmatic issues that have to be solved which are irrelevant to the purpose of this chapter. The specification of Pico$^-$ consists of eleven modules defining both syntax and se-

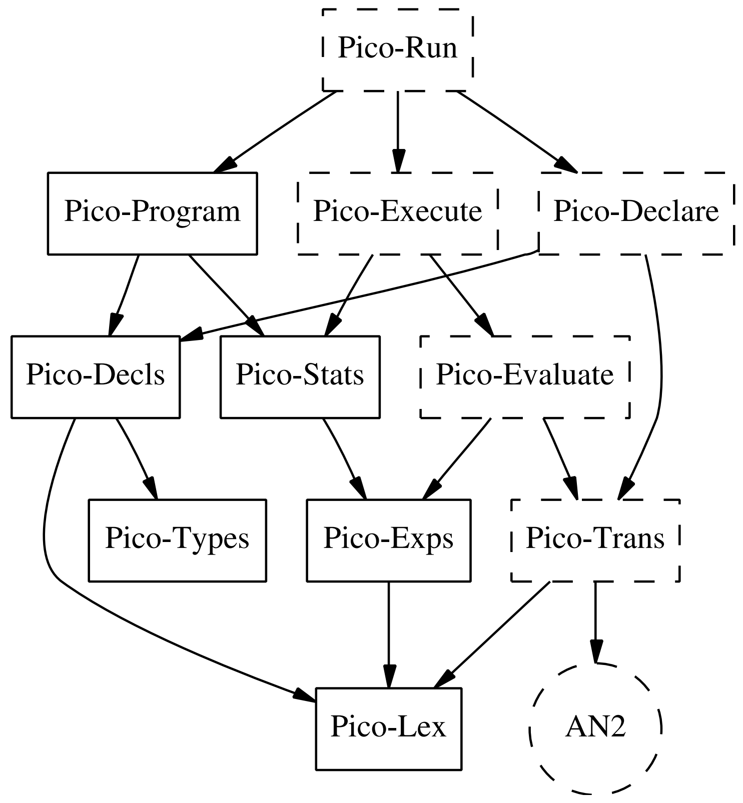


Figure 2.1: Import structure of the PICO$^-$ specification

mantics. They are displayed in Figure 2.1. The solid boxes form the syntactic modules, while the dashed boxes represent the modules containing the semantic equations. The circled node AN2 represents all modules of `evalan`, the action interpreter. Module Pico-Trans defines functions which translate various lowlevel PICO$^-$-sorts (integers, identifiers) to their action semantic counterparts.

Note that the definition of PICO$^-$ is not composed in the way as advocated in [14]. In that approach, there is one module for each syntactic construct, semantic entity, semantic function and semantic equation. The high degree of modularity thus obtained allows programming languages to be defined just by composing the appropriate modules. Although reuse of specifications has many benefits, it is irrelevant for the purpose of this chapter. We have therefore chosen to divide the definition of PICO$^-$ according to the different facets of the language (lexical, functional, imperative and declarative).

In the following, the syntax of PICO$^-$ will be clear from the semantic equations that will be discussed. Furthermore, uppercase identifiers occurring in the semantic equations denote variables.

2.3.2 Expressions

Expressions in PICO^{-} consist of integer expressions and identifiers referring to variables. For syntactic sorts of this kind a semantic function is declared in SDF:

```
"evaluate" "[[" EXP "]]" -> Action
```

This function maps any (PICO^{-}) expression to the sort containing all actions. There are three kinds of expressions in PICO^{-}: constant integer expressions, variable references and arithmetic expressions. The semantics of the first kind is the simplest: just provide the constant after translating it to an AN2 integer.

```
[ev-n] evaluate[[Int]] = provide integer of[[Int]]
```

Thus, evaluating a constant integer n means the same as the action *provide n*. The translation function `integer of[[]]` is defined in the aforementioned module Pico-Trans. It can be seen as a casting operator. There is one other casting operator, `token of[[]]`, which maps PICO^{-}-identifiers to AN2 tokens.

The second kind of expressions are variable references. Their semantics is defined as follows:

```
[ev-i] evaluate[[Id]] = inspect the cell bound to
                                              token of[[Id]]
```

The meaning of the right hand side of this equation is perfectly clear due to the use of Full AN2 syntax. The same action reduced to AN-K looks like:

```
give current bindings and provide token of[[Id]] then
      give bound_ then give the cell_ then inspect
```

Action combinators all associate to the left, so there is no need for any parentheses in this action. The first combinator is ***and***. An action A_1 ***and*** A_2 executes A_1 and A_2 independently, and concatenates their results if they both terminate normally. The other combinator is ***then***, which is used to pass data. For an action A_1 ***then*** A_2, the given data of A_1 is taken by action A_2. So, operationally, this action reads: get the active set of bindings, and provide token `id`, then retrieve the bound value, check that the given data is a cell and finally retrieve the value stored at the given cell.

The last kind of expressions are arithmetic expressions. We only describe the action semantics of $+$ since (unsurprisingly) the semantics of $-$ looks quite the same.

```
[ev-p] evaluate[[Exp1 + Exp2]] =
         evaluate[[ Exp1 ]] and evaluate[[ Exp2 ]] then
           give (the int #1 + the int #2)
```

In this equation, the denotational character of action semantics becomes apparent. The semantics of a composite expression is totally defined in terms of the semantics of its parts. The part after ***then*** is again Full AN2 syntax. It merely checks that the given data is a tuple of two integers and then performs the addition.

2.3.3 Statements

As opposed to expressions, statements are executed instead of evaluated. This motivates a new semantic function:

```
"execute" "[[" SERIES "]]" -> Action
```

For convenience this function is defined on series of one or more statements. Statements include assignment, if-then-else and while. Since booleans are absent from PICO$^-$ we need a function to guard if- and while-statements:

```
"truth" "of" "[[" EXP "]]" -> Action
```

This function maps (integer) expressions to actions which evaluate to either true or false.

The simplest statement is the assignment statement. Its semantics is defined as:

```
[ex-as] execute[[Id := Exp]] =
          give the cell bound to token of[[Id]] and
            evaluate[[Exp]] then update
```

The kernel action constant *update* takes cell-value tuple and replaces the content of the cell with the given value.

To execute an if-statement the guard expression is first converted to a truth value. Depending on this value the appropriate series of statements is executed.

```
[ex-if] execute[[if Exp then Series1 else Series2 fi]] =
          truth of[[Exp]] then infallibly select (
            (given true then execute[[Series1]]) or
            (given false then execute[[Series2]]))
```

The action prefix ***infallibly*** is a Full AN construct which ensures that the argument action will never fail. That is, if the select action should fail, an exception is raised. Although a ***select*** (A_1 ***or***...***or*** A_n) action denotes non-deterministic choice, the use of the two *given* guards makes it into a deterministic choice.

The semantics of the while construct looks very similar, except that the choice action is surrounded by ***unfolding*** and the succeeding branch continues with $unfold$ to reenact the loop.

```
[ex-wh] execute[[while Exp do Series od]] =
          unfolding (
            truth of[[Exp]] then infallibly select (
              (given true then execute[[Series]]
                 and then unfold) or
              (given false then skip))
          )
```

The ***and then*** combinator is used for sequential composition to make this an *imperative* loop. The Full AN2 constructs ***unfolding*** and $unfold$ have an interesting reduction semantics.

```
unfolding(A) =  give current bindings and
                  (provide ("unf", A) then
                      give binding_) then
                  give overriding_ hence A
```

This kernel action overrides the current bindings (locally) with a binding from the special token `unf` to the argument action A and then executes A itself in the context of these new bindings. The reduced version of *unfold* is slightly

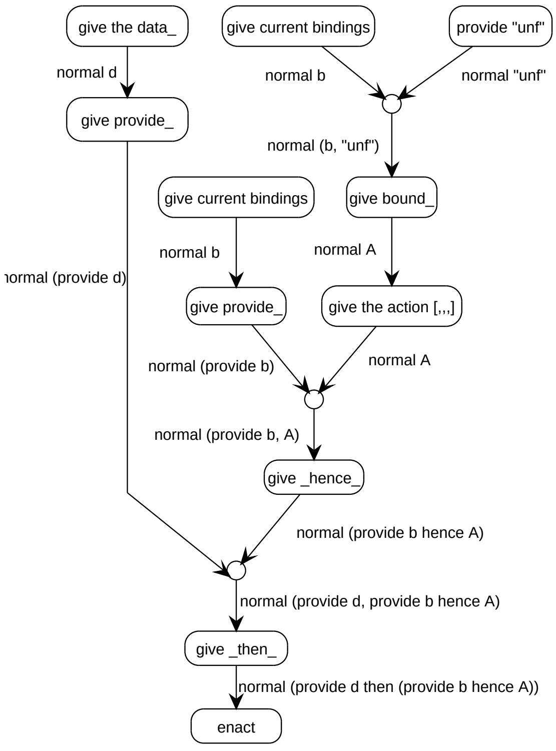


Figure 2.2: Data Flow Diagram for *unfold*

more complex. A new action is gradually constructed using reflection. This constructed action contains the original action bound to token `unf`.

```
unfold = give the data_ then
   give provide_ and
   (give current bindings then
       give provide_ and
       (give current bindings and
           provide "unf" then
           give bound_ then
           give the action[taking () giving bindings]_) then
       give _hence_) then
   give _then_ then
   enact
```

Let A be the unfolding action containing *unfold*, d the data given to *unfold* and b the active bindings. The action that eventually gets enacted is built up as displayed in Figure 2.2. In the end, it is ***unfolding***(A) that is enacted again, this time with other data given to it.

To complete the semantical description of statements, a definition for a series of statements is needed:

```
[ex-sq] execute[[Stat ; Stat+]] =
          execute[[Stat]] and then execute[[Stat+]]
```

Since statements do not return values, the sequential composition combinator ***and then*** is used.

2.3.4 Declarations

Declarations in PICO$^-$ consist of an identifier and a type (`natural`) and are located at the beginning of a program. There is thus only one global scope. The following functions are used to map declarations to actions:

```
"declare" "[[" DECLS "]]" -> Action
"declare" "[[" ID-TYPES "]]" -> Action
```

The sort `ID-TYPES` contains lists of terms of the form *id* : *type* separated by commas.

```
[dec1] declare[[declare Id-Types;]] = declare[[Id-Types]]
[dec2] declare[[ ]] = provide no bindings
[dec3] declare[[Id : Type, Id-Types]] =
         provide nothing then create
           then bind (token of[[Id]], the cell)
            and declare[[Id-Types]] then give disjoint union
```

The first equation defines the semantics of a PICO$^-$ declaration section to be just the same as the semantics of the declarations contained in it. Equation `[dec2]` is the base case of the recursion over lists of *Id : Type* pairs. The real work is done in the last equation. First the special data constant *nothing* is provided, indicating that the variable *Id* is uninitialised. Then a cell is allocated containing this value. The cell given by *create* is bound to the token of *Id*. Finally, the resulting binding is disjointly united with the bindings produced by the tail of the list. Note that no scoping constructs are used; the only concept we are dealing with here are bindings *as data*. Should the single scope character of PICO$^-$ change, these equations would not have to be modified.

2.3.5 Programs

The scope of declared variables is determined at the level of the program. The semantic function for programs combines the functions for statements and declarations. It is declared as:

```
"run" "[[" PROGRAM "]]" -> Action
```

There is only one simple equation for this function:

```
[run] run[[begin Decls Series end]] =
        declare[[Decls]] hence execute[[Series]]
```

The action combinator ***hence*** makes the bindings given by `declare[[Decls]]` available to the body of the program.

2.4 Example Pico$^-$ Program

Now that an action semantic description of Pico$^-$ is available, we can translate Pico$^-$ programs to actions. Consider the following program P, which computes the 5th Fibonacci number in j:

```
begin
  declare
    n: natural,
    i: natural,
    j: natural,
    k: natural;
  n := 5; i := 1;
  j := 0; k := 0;
  while n - k do
    j := i + j;
    i := j - i;
    k := k + 1
  od
end
```

The evaluation of `run[[`P`]]` produces the following action:

*provide ”nothing” **then** create **then** (provide ”n” **and** give the cell_ **then** give binding_) **and** (provide ”nothing” **then** create **then** (provide ”i” **and** give the cell_ **then** give binding_) **and** (provide ”nothing” **then** create **then** (provide ”j” **and** give the cell_ **then** give binding_) **and** (provide ”nothing” **then** create **then** (provide ”k” **and** give the cell_ **then** give binding_) **and** provide no bindings **then** give disjoint union_) **then** give disjoint union_) **then** give disjoint union_) **then** give disjoint union_ **hence** (give current bindings **and** provide ”n” **then** give bound_ **then** give the cell_ **and** provide 5 **then** update **and then** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **and** provide 1 **then** update **and then** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **and** provide 0 **then** update **and then** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **and** provide 0 **then** update **and then** (give current bindings **and** (provide (”unf”, give current bindings **and** provide ”n” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _−_) **then** (give the int_ **and** provide 0 **then** (check _>_ exceptionally fail **and** copy)) **then** provide true otherwise provide false **then** (select (give the data_ **then** give tupleToList_ **and** (provide true **then** give tupleToList_) **then** (check _=_ exceptionally fail **and** copy) **then** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _+_)) **then** update **and then** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and*** (give #2_ **then** give the int_) **then** give _−_)) **then** update **and then** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **then** inspect **and** provide 1 **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _+_)) **then** update))) **and then** (give the data_ **then** give provide_ **and** (give current bindings **then** give provide_ **and** (give current bindings **and** provide ”unf” **then** give bound_ **then** give the action[taking () giving bindings]_) **then** give _**hence**_) **then** give _then_ **then** enact) or give the data_ **then** give tupleToList_ **and** (provide false **then** give tupleToList_) **then** (check _=_ exceptionally fail **and** copy) **then** provide ()) otherwise (provide () **then** raise))) **then** give binding_) **then** give overriding_ **hence** (give current bindings **and** provide ”n” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _−_) **then** (give the int_ **and** provide 0 **then** (check _>_ exceptionally fail **and** copy)) **then** provide true otherwise provide false **then** (select (give the data_ **then** give tupleToList_ **and** (provide true **then** give tupleToList_) **then** (check _=_ exceptionally fail **and** copy) **then** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _+_)) **then** update **and then** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”j” **then** give bound_ **then** give the cell_ **then** inspect **and** (give current bindings **and** provide ”i” **then** give bound_ **then** give the cell_ **then** inspect) **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _−_)) **then** update **and then** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **and** (give current bindings **and** provide ”k” **then** give bound_ **then** give the cell_ **then** inspect **and** provide 1 **then** (give #1_ **then** give the int_ **and** (give #2_ **then** give the int_) **then** give _+_)) **then** update))) **and then** (give the data_ **then** give provide_ **and** (give current bindings **then** give provide_ **and** (give current bindings **and** provide ”unf” **then** give bound_ **then** give the action[taking () giving bindings]_) **then** give _**hence**_) **then** give _then_ **then** enact) or give the data_ **then** give tupleToList_ **and** (provide false **then** give tupleToList_) **then** (check _=_ exceptionally fail **and** copy) **then** provide ()) otherwise (provide () **then** raise)))))))

How to execute this action is surveyed in the next chapter.

Chapter 3

AN2 Tools: Overview

An earlier version of this chapter has been published as [40] *in* [34].

3.1 Introduction

Action Semantics is a formal, readable and modular formalism for the definition of programming languages. As such it greatly improves the maintainability and reuse of language definitions. However, given such pragmatic advantages over other semantic formalisms, we would like to be able to *use* the languages defined using action semantics. In this chapter we explore strategies for executing actions. These strategies comprise: execution by term rewriting, compilation to Java and C, and using an intermediate language. We have focussed on the following issues:

Generality The tools to execute actions should support the full kernel of AN2. This includes the interacting facet as well as reflection.

Self-containment Compiled actions should be a self-contained black box which can be subject to reflection (as provided by AN2).

Deployability It should be an easy job to embed actions, either interpreted or compiled, into existing software environments.

Extendibility Users should be able to extend interpreters or specialize compiled actions to their needs. Furthermore, the specifications of the compilers and interpreters themselves should be maintainable enough to allow (future) extensions or changes.

Portability There should be no restriction as to the platforms that are supported by the interpreters and compilers.

Efficiency Although not a primary goal, the performance of executing actions should be reasonable.

From these desiderata one can deduce that we deem the software engineering aspects of executing actions most important. This is a natural consequence of our view that action semantics is not only a formalism to define the mathematical semantics of programming languages, but also a language to define domain specific languages.

Organization First we give a very short introduction to the new Action Notation and discuss some distinctive features. In Section 3.3 we describe the term rewriting strategy which is divided in two parts. First we use the ASF+SDF Meta-Environment to derive a term rewriting system from the Modular SOS definition of the kernel of AN2 (AN-K). The second part discusses the pros and cons of reimplementing this term rewriting system by hand in C. In the next section we discuss a way to compile actions to C and Java. Section 3.5 describes the intermediate language approach. We conclude with assessment of the strategies presented, and a discussion of related work.

3.2 Action Semantics for Dummies

In this section[1] we give a short introduction to the new Action Notation [24, 32].

The central concept of action semantics is the action. An action is a computational entity that takes tuples of data and gives tuples of data. Actions can be primitive (e.g. computing a data operation, updating a cell) or combined using combinators that capture the various flows of data and control. For example: the **then** combinator is used for functional composition. This means that in action A_1 **then** A_2 the data given by A_1 is passed as input to A_2. Other combinators exist for sequential composition, interleaved composition, non-deterministic choice etc. Actions are able to terminate in three ways: normal, exceptional or failing. Normal and exceptional termination is accompanied by a data value (*given* resp. *raised* data). Combinators such as **exceptionally** and **otherwise** can be used to trap non-normal termination. For example: the action *raise* **exceptionally** *provide* () will terminate normally and give the empty tuple as result.

Action Semantics is divided over a number of so called *facets* which capture different ways of information processing. For instance, the functional facet consists of all actions having to do with information flow without side effects. Side effects are covered in the imperative facet. One of the distinctive differences between the AN1 and AN2 is that for the latter the facet responsible for the flow of bindings is included in the functional facet. That is, bindings have become a subsort of Datum (the sort of individual values) and can be processed as such by actions. There is only one basic combinator that deals with bindings: **hence**. Consider the action A_1 **hence** A_2. If A_1 terminates normally with bindings as a result, these bindings become available in A_2. The primitive action *give current bindings* returns the current set of bindings as data. It is then possible to obtain bound values using specific data operations.

Although facets were present in AN1, AN2 additionally distinguishes two levels of notation: Full AN2 and Kernel AN2 (AN-K). The semantics of Full AN2 is defined in terms of AN-K. As a consequence, AN-K is the only level tool support has to deal with to obtain full generality. Many familiar constructs from AN1 are now defined in terms of AN-K actions using reflection. Reflection in this case amounts to the dynamic *construction* of actions using combinators, as opposed to *inspection* of actions (like reflection in Java). The concept is simple, yet very powerful. The idea is that the sort of actions is subsort of Datum and consequently all action combinators are data operations. Consider for example

[1]This section repeats material from Chapter 2.

```
give current bindings and (provide ("unf", copy and provide 0 then (check
_=_ exceptionally fail) then provide 1 otherwise (copy and provide 1 then
(check _=_ exceptionally fail) then provide 1) otherwise (copy and provide
2 then give _-_ then (give the data_ then give provide_ and (give current
bindings then give provide_ and (give current bindings and provide "unf"
then give bound_ then give the action [taking () giving bindings]_) then give
_hence_) then give _then_ then enact) and (copy and provide 1 then give _-_
then (give the data_ then give provide_ and (give current bindings then give
provide_ and (give current bindings and provide "unf" then give bound_ then
give the action [taking () giving bindings]_) then give _hence_) then give
_then_ then enact)) then give _+_)) then give binding_) then give overriding_
hence (copy and provide 0 then (check _=_ exceptionally fail) then provide 1
otherwise (copy and provide 1 then (check _=_ exceptionally fail) then provide
1) otherwise (copy and provide 2 then give _-_ then (give the data_ then give
provide_ and (give current bindings then give provide_ and (give current
bindings and provide "unf" then give bound_ then give the action [taking ()
giving bindings]_) then give _hence_) then give _then_ then enact) and (copy
and provide 1 then give _-_ then (give the data_ then give provide_ and (give
current bindings then give provide_ and (give current bindings and provide
"unf" then give bound_ then give the action [taking () giving bindings]_) then
give _hence_) then give _then_ then enact)) then give _+_))
```

Figure 3.1: Recursive Fibonacci in AN-K

the following Full AN2 action which computes the Fibonacci number for a given integer:

```
unfolding(
  (copy and provide 0 then (check _=_ exceptionally fail) then provide 1)
  otherwise
  (copy and provide 1 then (check _=_ exceptionally fail) then provide 1)
  otherwise
  ((copy and provide 2 then give _-_ then unfold) and
   (copy and provide 1 then give _-_ then unfold) then give _+_)
)
```

The same action reduced to AN-K is displayed in Figure 3.1. Note how the behaviour of **unfolding** and *unfold* is mimicked by binding the special token "unf" and employing reflection to inline the unfolded body.

We will use the AN-K action displayed in Figure 3.1 to assess the runtime performance of the tools presented here as a running example. Furthermore, it gives a good impression of what kind of input we are dealing with here. As we require full support of the kernel of AN2, all tools presented here are able to execute this action in its kernel form without any knowledge of the higher level constructs behind it.

3.3 Execution by Term Rewriting

In this section we will review two term rewriting approaches to the problem of executing actions. In the first approach we used the ASF+SDF-formalism [23, 5] to specify the semantics for the kernel using conditional equations which are very close to the Modular SOS transitions of AN2 [32]. An interpreter is obtained by viewing these equations as rewrite rules. The second approach is a reimplementation in C of the derived term rewriting system.

In Figure 3.2 the interactions between the various translators and interpreters are displayed using T-diagrams [15]. The primary input is a program P written in language L. This program is translated to a kernel action by some specification in ASF+SDF. The resulting kernel action can be executed directly

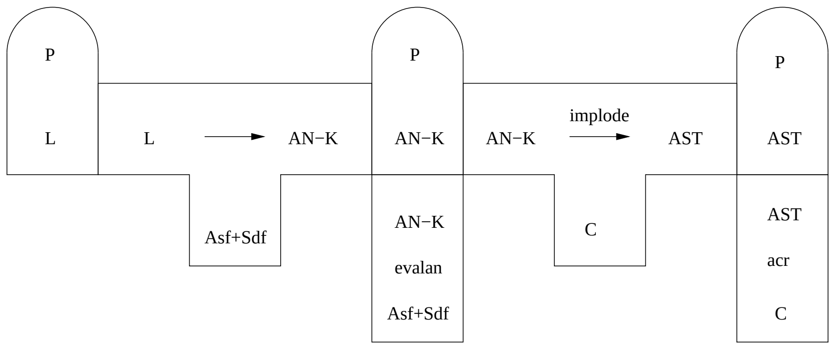


Figure 3.2: Interaction of term rewriting tools

by `evalan`, the Action Notation interpreter implemented using ASF+SDF. Secondly, an implosion step converts an AN-K parse tree to an abstract syntax tree (AST)[2]. The resulting AST can then be executed by the action rewriter `acr` which is implemented in C.

3.3.1 Kernel AN2 in Asf+Sdf

Since the semantics of Kernel AN2 is defined operationally, ASF+SDF is a useful tool for specifying it. The marriage of SDF to ASF enables one to write conditional equations on terms using concrete syntax. Directing these equations gives rise to a (derived) term rewriting system, which can be compiled to an efficient standalone tool. The interpreter `evalan` is such a specification. The specification of `evalan` consists of roughly eighty modules defining both syntax and semantics of data and actions. To actually see how the Modular SOS transition relations are translated to ASF+SDF, let's compare one of the relations defining the **then** combinator in CASL and ASF+SDF. If the left hand operand to **then** has terminated normally, the following transition rule applies:

$$\frac{\alpha' = \alpha[d_1/data] \wedge A_2 \xrightarrow{\alpha'} A_2'}{normal\ d_1\ \textbf{then}\ A_2 \xrightarrow{\alpha} normal\ d_1\ \textbf{then}\ A_2'}$$

The transition declares that d_1 is known during the one-step execution of A_2 in the *data* field of label α' . In ASF+SDF this transition is defined in one (conditional) equation:

[2]This step is provided for by a tool accompanying ASF+SDF: `implodePT`. Since this tool is provided as is, the implementation language (C) is in fact irrelevant.

```
[sos4] s' = s[d1/data],
       s' |- A2 = <A2', s''>
       ===
       s |- normal d1 then A2 = <normal d1 then A2', s/s''>
```

Since ASF+SDF has no notion of truth *and* we want our specification to be executable, assertions are replaced with equations that faithfully represent the intended semantics of the CASL specification. The concept of a Modular SOS label (an arrow of a category) is mimicked by passing a state (an object of a category) throughout the evaluation of $\cdot \vdash \cdot$ and an operator $\cdot/\cdot$ which prohibits the observation of local changes. The equation assigns a new state s' updated with d_1 in the *data* field of s which is passed to the one step execution of A_2. This results in a residual and a (possibly modified) state s''. The result of the equation is a tuple of the original action with A_2' substituted for A_2 and the observable fields of the new state.

The transitive closure of the one step equations is defined as usual. Since we use the $\cdot/\cdot$ operator in appropriate places, bindings and transients are local to one step execution equations. The transitive closure thus only allows observable information to be observed.

The interpretation function to perform an action A with input d is now defined as follows:

$$\text{perform}(A, d) = \varepsilon \vdash^{+} \textit{normal } d \textbf{ then } (\textit{normal no bindings} \textbf{ hence } A)$$

In this equation ε denotes the initial state.

3.3.2 Action Rewriting in C

Although ASF+SDF provides a useful framework for specifying the semantics of Kernel AN there are some drawbacks. Since term rewriting is the only computational device ASF+SDF (currently) supports, even the most primitive operations are to be specified using equations. For instance addition of two integers is computed in a term rewriting fashion. Of course we would like to use native support to compute integer operations to speed up execution. The same holds for updating of stores, rotating the schedule and so on. Furthermore, ASF+-SDF's capabilities seemed far too general for our purposes: since signatures are described by production rules in SDF we had to devise *concrete* syntax for all auxiliary structures such as stores, bindings, finite maps etc. Disambiguation of all these sorts—using syntactic sugar—hindered a perspicuous implementation even more. Finally, we thought we could improve upon the performance of `evalan`. These considerations taken together, led to the decision to reimplement the term rewriting system by hand in C. Having a valuable prototype in ASF+SDF and the ATerm library [7] to implement a data notation, this was not a hard job.

`Acr` takes an Abstract Syntax Tree as input. The algorithm traverses the tree in a Modular SOS fashion: the traversal is directed by combinators, until a subtree can be collapsed. The tree thus decreases in depth in a bottom up fashion, while traversing it top down for each step. The main difference between `acr` and `evalan` is that `acr` employs term rewriting only to reduce combined actions. All primitive actions (including data operations and predicates) are hardcoded in C. Especially for interacting actions the performance gain is expected to be more than marginal since no matching is involved in rotating the schedule (and this is likely to occur very often).

3.3.3 Comparing `evalan` and `acr`

Since both `acr` and `evalan` traverse the action tree for each computational step, the complexity of the reduction algorithm should be roughly the same. However, primitive actions such as updating a cell have complexity $O(1)$ in `acr`. The complexity of primitive actions in `evalan` depends on the data structures involved. For example, updating a cell in `evalan` will cost $O(n)$ in the worst case, where n is the number of allocated cells. This is due to the fact that finite maps in ASF+SDF are essentially lists of tuples. To assess the influence of these primitive actions more concretely, we compared the performance the of the Fibonacci action presented earlier and an iterative version. The results show that `acr` is on average two times as fast as `evalan` for the recursive algorithm. Since no imperative actions are used and bindings are implemented using bounded balanced trees [1] both in `acr` and `evalan` this can only be accounted for by the arithmetic operations and the more complex matching of terms in `evalan`. For the iterative Fibonacci algorithm we see that for larger values of n (≈ 100) `evalan` takes considerably more time than `acr` and this difference is growing fast. Again term rewriting of arithmetic is probably the cause of this.

Remark The fact that `acr` traverses the tree and constructs new ones on the way up for each step, could have been avoided by using a different tree representation. (Abstract) Syntax trees resulting from ASF+SDF related components are always represented by ATerms which allow no destructive updates (copy-on-write semantics). Using a tree representation that would allow destructive updates on subterms, an executing agent could have been represented by a *cursor* walking over the tree, collapsing and creating trees only locally. However, there is one 'small' problem to this approach: syntax trees would need an enormous amount of memory compared to the corresponding ATerm, since ATerms are maximally shared. Precisely this problem of explosion in size has been one of the reasons for making ATerms maximally shared [7].

Assessment While experiments show that `acr` is generally faster than `evalan`, the handcoded approach has a number of disadvantages. First, `acr` uses a fixed length representation for integers. Thus, integer values are restricted to the size native to the machine `acr` is running on. A related problem is that the used data notation of `acr` is hard to extend by the user. For `evalan` one has the full algebraic power of ASF+SDF available to extend the interpreter with arbitrary data types. This is achieved using an interface mechanism. If a user specifies his own data constructors and data operations one can extend the interpreter by complying to this interface and adding his equations to the equations of the `evalan` specification. Using the interface the interpreter can "know" about foreign data constructs. The meaning of the data operations is specified using the generic `result` function which is used by `evalan` to execute **`give`** *o* actions. Depending on the importance of full generality and extendibility it might not be such a good idea at all to reimplement the term rewriting system by hand. Moreover, `acr` has been designed to primarily optimize non-functional aspects such as scheduling and store updates, so for purely functional actions the performance penalty induced by `evalan` is expected to be relatively small and constant (as is corroborated by the comparison of recursive Fibonacci).

3.4 Compilation to Java and C

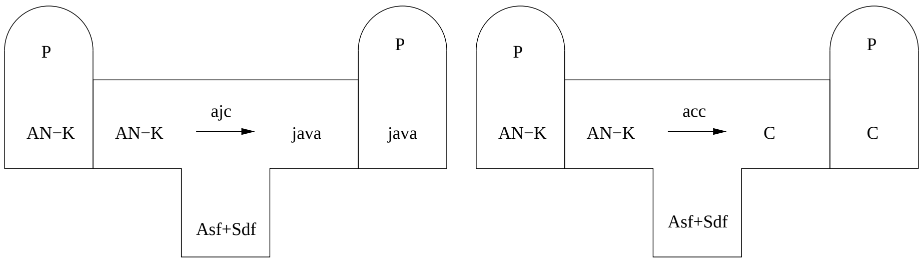


Figure 3.3: Compilation of AN-K to Java resp. C

In this section we describe ways to compile actions to Java and C. In Figure 3.3 the compilers `acc` and `ajc` are depicted using T-diagrams. Both compilers are implemented using ASF+SDF. As before, we only consider kernel notation, with the exception of unfoldings which are detected by `ajc`. For the compilation to C we restrict ourselves to single threaded actions. We require our compiled actions to be self-contained and compositional. Self-containment allows for easy deployment of actions, while compositionality ensures the possibility of (off line) reflection. To achieve these requirements, we introduce Action Functors in C and Enactables in Java. Both are interfaces (signatures) to which compiled actions should comply. Since we require that reflection is supported, this opens the way to separate compilation of actions (even from different source languages) and then combining them into one. Furthermore, in Java, an action implementing the Enactable interface can also implement the standard Serializable interface, which makes it possible to store actions on file or send it over a network connection.

First we describe `acc`, the action to C compiler. We then compare this compiler to `ajc`, the java compiler.

3.4.1 Action Functors

An Action Functor in C is a type definition describing a function that represents an action, that is, a function that transforms data and bindings into data, while perhaps referencing cells. Its definition in C is as follows:

typedef AN_Data ($*$ACCFunctor)(AN_Data,AN_Data);

The `acc` runtime library defines all Kernel AN2 primitive actions in such a way that they obey this function type. The compiler translates an action tree by mapping each subtree to an Action Functor. So for example A_{i0} then A_{i1} at position i in the tree is compiled to:

```
AN_Data action_i(AN_Data data, AN_Data bindings) {
  register AN_Data temp = action_i0(data, bindings);
  return action_i1(temp, bindings);
}
```

One would expect that this way of compiling actions to C would result in inefficient code, since the number of function calls is more or less equal to the number of nodes in the action tree. However, we have experimented with different compilation schemes (such as using intermediate variables in one big function, or exploiting the runtime stack) but they, suprisingly, all turned out to be slower than the compilation scheme presented here. This is probably due to sophisticated optimizations of GCC which break down in the latter cases.

To cope with exception handling and (non deterministic) choice we use a choice point library which allows non-local jumps at a very high level [26]. To illustrate this, the code for A_{i0} exceptionally A_{i1} looks like this:

```
AN_Data action_i(AN_Data data, AN_Data bindings) {
  if (!ACC_try())
    return action_i0(data, bindings);
  else {
    register AN_Data temp = ACC_catch_exception();
    return action_i1(temp, bindings);
  }
}
```

Here ACC_try returns 0 when setting the choice point and returns 1 when action$_{i0}$ raised an exception or failure. In the **else** branch ACC_catch_exception returns the raised data in case of an exception and rethrows a failure otherwise. When an exception or failure occurs all registers and local variables are restored.

Now, the hard part is, of course, reflection. To accomplish *real* reflection in a portable way, we employed Paolo Bonzini's GNU Lightning. Lightning has been designed to implement fast just-in-time compilers, and has been used as such in GNU Smalltalk. It provides a set of macros that define a generic assembly language. These macros allow a programmer to build ordinary C functions at runtime for a number of platforms (i386, Sparc and PowerPC). Data operations defined on action operands are implemented by this runtime assembler, specialized for Action Functors. Since any action may be operand to, e.g. _**then**_ we have to ensure that all actions present in the runtime have the function type defined earlier. Currently, `acc` does not yet detect unfoldings to prevent reflection at runtime, but a memotable prohibits the construction of an action for each pass through a loop.

3.4.2 Enactables

Enactables are the Java equivalent of Action Functors. Enactables are classes that implement the Enactable interface. This interface declares one method `enact`, accepting Data and Bindings and returning Data. Enactables can be embedded in Action classes which are subject to reflection. The compilation to Java proceeds much in the same way as for `acc`, except that `ajc` generates a class implementing the Enactable interface for the top action, and private methods for each subaction. Provided actions are compiled to an inner class implementing the Enactable interface. The action data operations receive Action instances that are constructed from Enactables. This way it is easy to combine compiled actions with hand written Java classes. Another difference is the implementation of the data notation. Since the subclass concept resembles the subsort relation in AN2, implementation of the basic data types was straightforward. Runtime type checks are performed using standard casting operators of Java. The Data Notation is implemented using the Factory design pattern [18] to allow future changes[3] to the representation of values.

3.4.3 Comparing `acc` and `ajc`

Since `acc` and `ajc` compile actions almost in the same way, the comparison in performance between C-compiled actions and Java-compiled actions does not say us much about the compiler, but more about the difference between C and Java as a target. Generally speaking, the recursive fibonacci action compiled to C executes approximately 2.5 times faster than the same action compiled to Java. For the iterative version this factor is close to 5.

Assessment It is obvious that actions compiled to C are more efficient than actions compiled to Java. However, the actions in Java have a number of important pragmatic advantages. It is our view that these advantages outweigh the performance penalty by far. This is motivated by the following observations:

- First of all the slogan "compile once, run anywhere" applies here.
- The use of the Enactable interface allows the combination of compiled actions with *arbitrary* Java code. This is also possible for actions compiled to C but the process is more tricky and less type safe.
- For extension or specialization of compiled actions one has the complete Java runtime library at one's disposal.
- Java classes are mobile. E.g. sending compiled actions over a network connection using serialization should pose no problem.
- Object Orientation alleviates the burden of changing and/or maintaining the `ajc` runtime library. Since the java runtime library contains a host of standard data structures the supported data notation can be extended uniformly.

3.5 Action Intermediate Language

It is evident that the way action trees are reduced in the term rewriting paradigm is a source of inefficiency. The compilation to C remedies most of this, but at the cost of full generality: multi threading is not supported. Another drawback is that compiled actions are not mobile in a dynamic way. Combination of two compiled actions always involves writing a glue function which uses the reflection

[3]For example, an implementation based on ATerms would allow communication between actions compiled to Java and actions compiled to C.

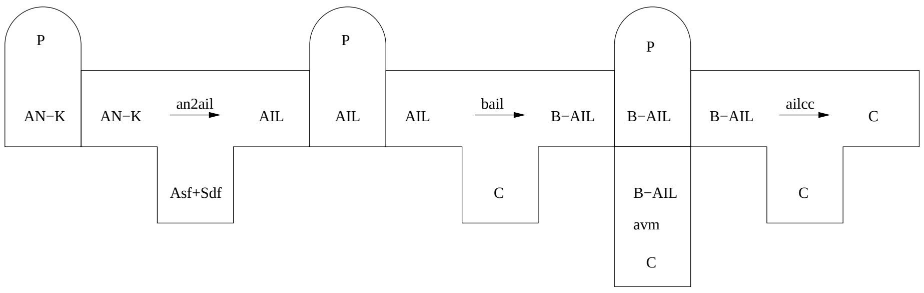


Figure 3.4: Compilation of Actions to AIL

operations of the `acc` runtime and (re)compiling the result. In this section we discuss the intermediate language approach to compilation of actions. Using Action Intermediate Language (AIL) we obtain full generality and performance comparable to that of actions compiled to C.

The interaction of the tools involved in the compilation of actions to AIL is depicted in Figure 3.4. We assume that *P* is available as a Kernel AN2 action. This action is first translated to an AIL parse tree (`an2ail`) which is then converted to a binary representation by the independent tool `bail`. The resulting object can be executed by the Action Virtual Machine (`avm`) or compiled to C.

Speed is achieved by translating the action tree to a sequence of instructions. The advantages are obvious. For example, in case of an exception, we can now jump to the appropriate handler instead of stepwise proliferating the exception up the action tree. One execution step now corresponds to incrementing an instruction pointer, instead of traversing a very large tree. Action Intermediate Language (AIL) has primarily been designed to remedy the efficiency problems of the term rewriting approach. Of course, since a number of SOS steps are collapsed into one jump, the behaviour of multi threaded actions can significantly differ. We expect, however, that it is possible to achieve correct operational behaviour modulo exceptional/alternative flow for multi threaded actions by inserting appropriate yield points in AIL byte code.

Action Intermediate Language can be seen as a typed assembly language of the stack based kind. AIL instructions can accept one (optional) parameter. If a parameter is present it must be an integer (indicating a non-datum constant value), a label, an ATerm (representing a datum or a data sort) or AIL code itself. The last argument type allows for reflection. Labels are translated to offsets by the bytecode compiler to ensure compositionality for AIL bytecode. Again, we use ATerms for the representation of data. One of the reasons to use ATerms for AIL bytecode lies in the efficient IO capabilities of the ATerm library. ATerms can be written to file *while retaining maximal sharing*. So we get serialization of all data (including actions provided as data) for free. The standalone tool `bail` takes an AIL parsetree as input and then builds an array of bytes, mapping instructions to opcodes and serializing data to binary ATerm format within the stream. The resulting byte stream is converted to a BLOB (Binary Large OBject) ATerm and written on file.

A part of the instruction set of AIL is displayed in Table 3.1. Data opera-

Aspect	Instructions
Given Data Flow	`prov, push, drop, copy, merge`
Raised Data Flow	`eprov, epush, edrop, ecopy, emerge`
Argument Data Flow	`publish, unpublish, epublish`
Scope Data Flow	`enter, leave, scope`
Normal Control Flow	`frame, goto, return, enact`
Escaping Control Flow	`trye, tryf, raise, throw, fail, catch`

Table 3.1: Subset of AIL Instructions

tions, predicates and primitive instructions more or less correspond directly to Kernel AN2 primitive actions and are not listed in the table. A difference is that typed primitive instructions do not check their arguments for type correctness. This is dealt with by the special instruction `cast`. The data flow instructions all operate on a number of stacks the function of which is explained in the next section. Control flow instructions are able to change the instruction pointer. The instruction `goto` l just does this. If a frame needs to be allocated, the instruction `frame` can be used. The label argument to this instruction denotes a return address. The instructions `trye` and `tryf` install a handler at the program point specified by the argument label.

3.5.1 AIL Runtime Environment

The runtime environment for AIL programs consists of a store, a schedule and a code region. When a program is run, an AIL Control Block (ACB) is created for agent “main”. ACBs contain all necessary data structures that are local to a thread (or agent). At runtime, new ACBs may be added to the schedule by the `activate` instruction. ACBs consist of two registers and a number of stacks. The first register is used for normal data flow (nreg) and the second for exceptional data flow (ereg). Each register has an associated stack (nresult resp. eresult) that is used for saving computed values. All primitive instructions *not* merely concerned with flow of data, take arguments from the registers and return values in the registers. If a sequence of instructions needs the same argument, the register is first saved on the argument stack and copied back into the register when needed. The third kind of stack is used for binding flow: the scope stack. Bindings can be transferred to and from the normal flow component. Finally, ACBs contain a context stack and a frame stack. The context stack is used for exception and failure handling. Contexts contain snapshots (shallow copies) of the result stacks, argument stack and scope stack, used for restoring the environment after an exception has occurred. Furthermore, a context contains a reference to the frame in which it was saved and a continuation address. The frame stack is used to save the return address when AIL code is enacted. To elucidate the way data may flow at runtime, the basic data flow instructions are depicted in Figure 3.5.

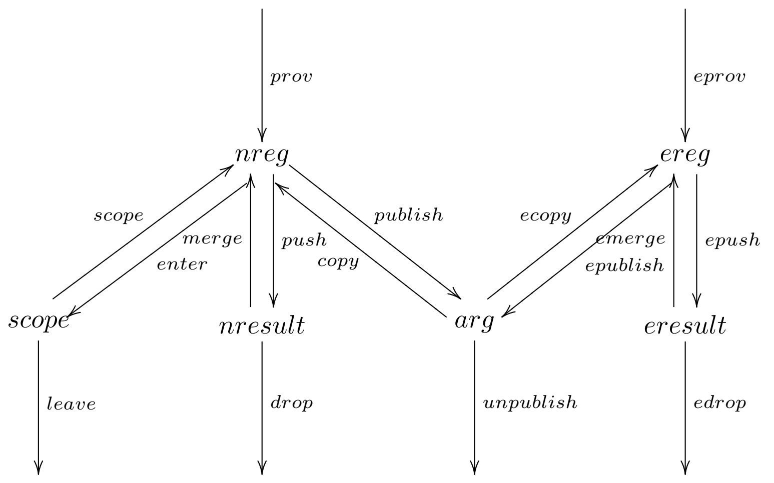


Figure 3.5: Transfer graph relating stacks and registers

3.5.2 Mapping AN-K to AIL

Let's look at how the basic action combinators are translated to AIL. The normal data flow and binding flow translations are rather obvious. The following four sequences represent the **then**, **and then**, and **hence** combinators (`[[A]]` denotes the translation of a sub action of a combinator).

```
[[A1]]
publish;
[[A2]]
unpublish;
```

```
[[A1]]
push;
copy;
[[A2]]
merge;
```

```
[[A1]]
cast bindings(<term>);
enter;
copy;
[[A2]]
leave;
```

In the first sequence A1 executes and leaves its result in nreg. This result is published onto the argument stack. Then A2 executes, probably using the published value. Finally the published data is withdrawn. In the second sequence, after A1 has finished the value in nreg is pushed onto the result stack. Then the published value (incoming data) is copied from the argument stack back into nreg. So, when A2 executes it receives the same input as A1. Finally the `merge` instruction prepends the top of the result stack to the data in nreg. The third sequence involves scoping. Since hence is the only typed combinator we have to ensure that the data given by A1 is of the proper type. The `cast` instruction leaves nreg as it is when the data is of the argument type, i.c. bindings, and throws an exception otherwise. If A1 returned bindings they are pushed onto the scope stack, making them the current set of bindings. Then the published data is copied and after A2 has finished, the current scope is left.

To see how exceptional control flow is mimicked in AIL, the translation of the **and exceptionally** combinator is an interesting case. An action A_1 **and exceptionally** A_2 terminates exceptionally if both subactions terminate exceptionally. This exception is accompanied by the concatenation of the raised data values of the subactions. In AIL this is achieved by guarding the righthand action with a `trye` block and throwing an exception with concatenated data in the handling part. Thus, A_1 **and exceptionally** A_2 is mapped to:

```
         trye l1;      // install handler at l1
           [[A1]]
         catch l2;     // branch to l2
    l1:    epush;      // push raised data onto eresult
           trye l3;    // install another handler
             [[A2]]
           catch l4;
    l3:      emerge;   // merge raised data
             throw;    // throw exception
    l4:    edrop;      // pop datum of first exception
    l2:    ...
```

Recall that `emerge` prepends the top of eresult to the data in ereg (the data raised by A_2). Note also that the `catch` instruction just pops the context stack and branches to the argument label.

The output of `an2ail` for the fibonacci action presented earlier is displayed in Figure 3.6[4]. Note that no reflection is present since **unfolding** and *unfold* are detected by the compiler.

3.5.3 Simple Optimization of AIL

The reason to separate the type checking from the actual computation of data operations is that if the type is statically known, the `cast` instruction can be left out. The use of registers in combination with stacks allows a number of simple optimizations of AIL code that reduce the number of stack operations considerably.

$$\begin{aligned}
\texttt{publish; } i\texttt{; unpublish;} &= i; \\
\texttt{update; push; copy; } i\texttt{; merge;} &= \texttt{update; copy; } i; \\
\texttt{copy; copy;} &= \texttt{copy;} \\
\texttt{copy; prov } \mathit{term}; &= \texttt{prov } \mathit{term};
\end{aligned}$$

The first rule states that if only instruction i uses the published data (which still resides in nreg after `publish`), the data need not be pushed onto the argument stack at all. In the second equation, the merging of data can be left out, since `update` (if terminating normally) returns the empty tuple. The third equation is obvious. Finally, for an instruction that never uses the data given to it, the published data does not have to be copied into nreg.

3.5.4 Assessing `avm`

Experiments with the recursive fibonacci action show that interpreting bytecode is slightly slower than the action compiled to C but this difference is growing

[4]Labels are preceded by an @ in this listing.

```
{
            frame @l2;
       @l1:
            tryf @l11;
            tryf @l101;
            copy;
            push;
            prov int(0);
            merge;
            publish;
            trye @l100002;
            cast [⟨appl(⟨term⟩)⟩,⟨appl(⟨term⟩)⟩];
            eq;
            catch @l100003;
  @l100002:
            fail;
  @l100003:
            unpublish;
            prov int(1);
            catch @l102;
     @l101:
            copy;
            push;
            prov int(1);
            merge;
            publish;
            trye @l100102;
            cast [⟨appl(⟨term⟩)⟩,⟨appl(⟨term⟩)⟩];
            eq;
            catch @l100103;
  @l100102:
            fail;
  @l100103:
            unpublish;
            prov int(1);
     @l102:
            catch @l12;
      @l11:
            copy;
            push;
            prov int(2);
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            sub;
            unpublish;
            publish;
            frame @l101001;
            goto @l1;
  @l101001:
            unpublish;
            push;
            copy;
            push;
            prov int(1);
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            sub;
            unpublish;
            publish;
            frame @l101011;
            goto @l1;
  @l101011:
            unpublish;
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            add;
            unpublish;
      @l12:
            return;
       @l2:
            return;
}
```

Figure 3.6: Recursive Fibonacci in AIL

for large values of n. We have not yet tested the additional compilation to C (`ailcc`), since this is not fully operational yet.

The pragmatic advantages of the intermediate language approach are obvious. Actions are compiled to a single object which is mobile, compositional and self contained. Offline combination of different actions is straightforward using the reflection primitives of AIL;—the result is just another binary object file. This is an improvement with respect to actions compiled to C where one has to recompile the separate compiled actions if one wants offline combination. Another pragmatic advantage is the extendibility of `avm`. Although this cannot be done dynamically or on request, `avm` is designed in such a way that extension consists only of adding instructions to the instruction set and defining their semantics in C. No changes to `bail` or `ailcc` have to be made.

3.6 Conclusions and Discussion

In this final section we compare our work to previously conducted research in this area and present some conclusions.

3.6.1 Discussion

The generation of interpreters and compilers from action semantic descriptions has a long history (cf. [8, 42, 38]). For the new Action Notation however, this field of research has only just begun. In this chapter we have presented some strategies for interpreting and compiling actions in ways that have not been explored before. Our approach differs in a number of aspects from the previously conducted research in this field.

The first and foremost difference is the central role of the algebraic specification formalism ASF+SDF. The Action Semantic Description (ASD) tools [42] used ASF+SDF to generate term rewriting systems from an ASD, a formalism on its own. That is, the ASD tools operated by first parsing an ASD and then *generating* a number of ASF+SDF modules which could be used to check and execute the language defined. In our approach, however, ASF+SDF *itself* is used as action semantic description formalism. Syntax is defined in SDF, which is then rewritten to action terms by the ASF component. This approach has a number of important advantages. First, since SDF is a declarative formalism that allows the full class of context-free grammars, the syntax of a language is easily specified and need not be molded into the class of LR or LALR grammars. This has the additional advantage that grammars can be designed in a modular way, since only full context-free grammars are closed under union. Furthermore, the compositionality of action notation and the algebraically defined data notation make ASF a perfect formalism for mapping syntax to semantics. This can be done using concrete syntax which makes action semantic descriptions all the more readable. Finally, a specification can easily be used outside the Meta-Environment by reusing the standalone components of ASF+SDF. In this paper we have described an interpreter of actions which is independent of the language defined: the interpreter is only defined for Kernel AN2. The composition of a language specification and the interpreter yields an interpreter for the language defined.

Previous efforts to execute actions were primarily focussed on performance issues relative to hand written compilers and interpreters. The execution speed of, e.g., OASIS [38], depends largely on thorough analysis of actions, involving for example binding time analysis, and on restricting the set of action combinators and primitives that are supported. While not disregarding the issue of performance, we take the opposite route, by stating that support for the full kernel of AN2 is the primary goal. This includes the interacting facet and reflection. Of course, we probably have to pay for this in terms of execution speed, but since bindings have become data and loops are defined using reflection in Kernel AN2, this may turn out to be unavoidable without the analysis of Full AN2 constructs. In the framework we have presented compilation and/or interpretation can always be preceeded by numerous analysis and optimization phases if this turns out to be desirable. A related design decision is to strive for portability. Therefore the option of compiling to native code has never even been considered.

A third difference with existing approaches is our focus on using actions in the real world. Put differently, we see action semantics not only as a formalism to mathematically define an programming language's semantics, but possibly also as a way to define Domain Specific Languages [10, 11, 12, 13]. This poses the question how to connect generated interpreters or compiled actions to existing

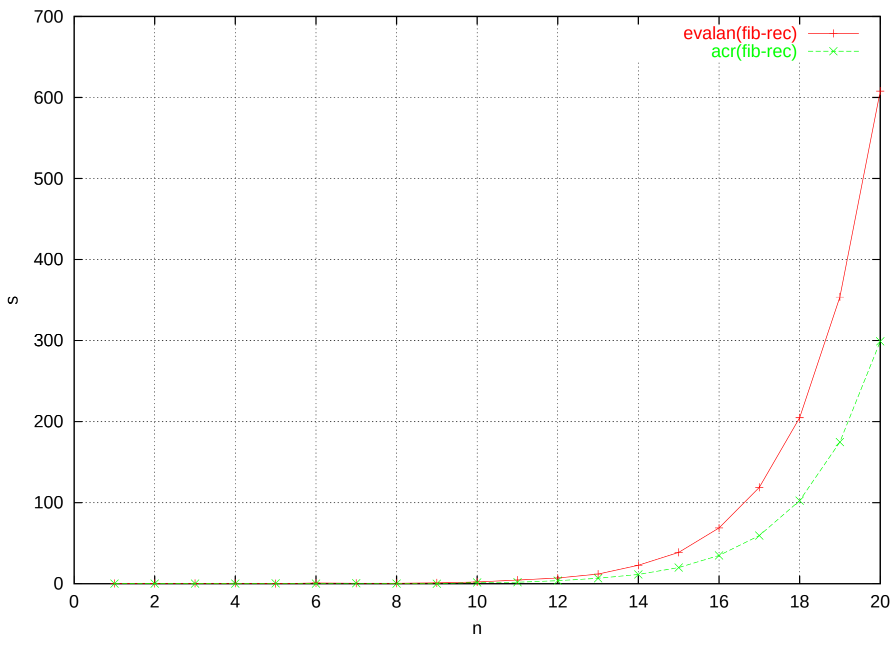


Figure 3.7: Execution times in seconds for Recursive Fibonacci executed using `evalan` and `acr`.

software environments. One solution to this has been provided by the ASF+-SDF Meta-Environment itself: compiled specifications can be connected to the Toolbus coordination architecture [3]. A second solution, addressed in this paper, is compilation of actions to a Java class definition. By letting this class implement the Enactable interface, a compiled action can be combined with handcoded "enactables" using reflection as provided by AN2.

3.6.2 Conclusions

We have presented a number of interpreters and compilers to put action semantic descriptions to use. From the experiments we have performed some conclusions can be drawn. First of all, if performance is important, the term rewriting approaches will not do. The difference in execution time between the term rewriting approach and the compiled actions (C, Java and AIL) is indeed dramatic. This conclusion can be drawn immediately from the plots displayed in Figure 3.7 and Figure 3.8[5]. The reason why one would still want to use `evalan` lies in the fact that it can be augmented with arbitrary algebraic data types. Additionally, the tight integration with the ASF+SDF Meta-Environment allows for the interactive specification as well as testing of programming languages. The interpreter `evalan` might best be used during the process of designing a language.

Compiling actions has numerous advantages over interpretation. The resulting objects are faster and easier to embed in existing software environments.

[5]All tests have been executed on an AMD Athlon XP1800+ running Linux.

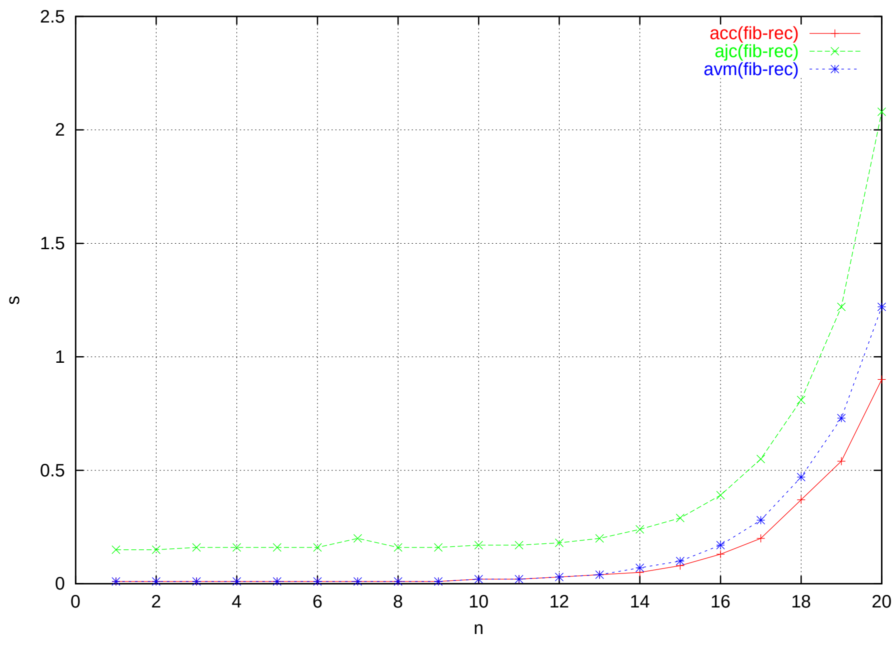


Figure 3.8: Execution times in seconds for Recursive Fibonacci executed using `acc`, `ajc` and `avm`

The compilers can be instructed to map certain (primitive) actions to user provided native code. This way it is possible to let actions really connect to the real world. We conclude that, if performance is not the main objective, the compilation to Java has the best credits for defining domain specific languages, since it combines Object Orientation with deployability and mobility.

3.6.3 Future Work

First of all we will have to test the tools presented here with interacting actions as input. Since we have only assessed the performance of a very small action in this paper, the question whether the interpreters and compilers will scale up to larger actions is an urgent one. We are currently working on the migration of the JOOS action semantics [44] to Asf+Sdf.

Although it is explicitly not our intention to compete with commercial compilers, more benchmarking should enable us to assess the performance penalties of the various strategies more accurately. There is reason to assume that there is room for improvement, especially for the compilers. Restricting the compositionality requirement to apply only to the top action is an option we will have to explore. But this is probably hardly possible without a thorough analysis of actions. These analyses should take both Full AN2 constructs and Kernel AN2 constructs into account. The problem is, that, whereas it is easier to analyse Full AN2 constructs only, a user is still permitted to use Kernel AN2 in his language definitions. As a consequence any agressive optimization that does not take both levels into account may lead to loss of information. For example, algebraic simplification of Kernel AN2 actions might destroy the link between some kernel subactions and their Full AN2 origins (e.g. **unfolding**). An important step to more efficient compilation (especially to C and Java) would be the availability of a staticness condition. It would then be possible to eliminate all bindings and possibly many type checks.

Apart from the issue of performance, we plan to center our future work around increased usability and deployability of actions. This work includes making the interpreters as well as compilers extendible, connecting all tools to the Toolbus coordination architecture [3], and targeting the .NET intermediate language [25].

Chapter 4

Rewriting Actions

4.1 Introduction

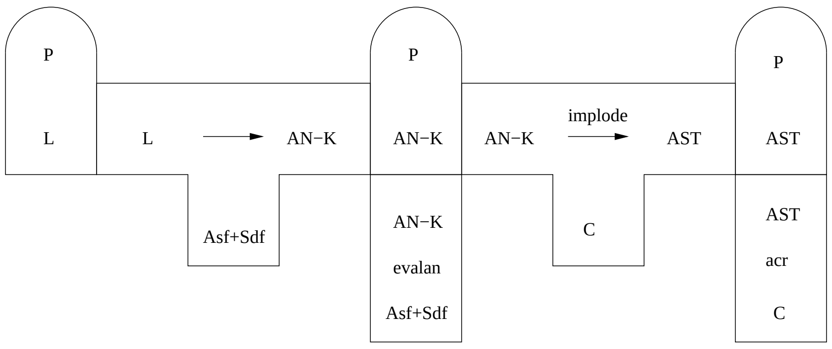


Figure 4.1: Interaction of term rewriting tools

This chapter is devoted to exploring the links between the Modular SOS (MSOS) semantics of Kernel AN2 (AN-K) and the interpreters which use term rewriting to execute actions. To recapture, the T-Diagram from Chapter 3 is displayed in Figure 4.1. It is assumed there exists a translation from a language L to a term over the kernel of AN2 (AN-K). First, this term can be executed directly by `evalan`, the interpreter defined in ASF+SDF. Second, the action can be imploded to an Abstract Syntax Tree (AST) and then be executed by `acr`, the action rewriter implemented in C. The chapter is organized as follows. First we present a short introduction to Modular SOS and comment on how this can be implemented in ASF+SDF. Then some issues of modularity and data notation are discussed. Finally we describe `acr` and propose a solution to the problem of extending `acr`.

4.2 Implementing MSOS in Asf+Sdf

4.2.1 Crash Course in Modular SOS

In Modular SOS (MSOS) all state information is located on the labels of a labeled transition system. That is, a label determines the state before *and* after a transition. The structure of a label is a that of category. We take the definition of a category from [31]:

Definition 1 *A category consists of a set of arrows $\alpha \in \mathbb{A}$ and a set of objects $o \in |\mathbb{A}|$, together with total operations $pre, post : \mathbb{A} \to |\mathbb{A}|$, $id : |\mathbb{A}| \to \mathbb{A}$, and a partial composition operation $\cdot\,;\cdot : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, such that:*

- *$\alpha_1;\alpha_2$ is defined iff $post(\alpha_1) = pre(\alpha_2)$, and then $pre(\alpha_1;\alpha_2) = pre(\alpha_1)$ and $post(\alpha_1;\alpha_2) = post(\alpha_2)$;*
- *$\cdot\,;\cdot$ is associative, that is $\alpha_1;(\alpha_2;\alpha_3) = (\alpha_1;\alpha_2);\alpha_3$ when defined;*
- *$id(pre(\alpha));\alpha = \alpha = \alpha;id(post(\alpha))$;*
- *$pre(id(o)) = o = post(id(o))$.*

The operations $pre(\alpha)$ and $post(\alpha)$ designate the *source* and *target* of arrow α respectively. The set of identity arrows $id(o)$, which is a subset of $\mathbb{A}$, is designated by $\mathbb{I}^{\mathbb{A}}$ or $\mathbb{I}$ if $\mathbb{A}$ is clear from the context. We let variables ι range over $\mathbb{I}$, and variables α range over arbitrary categories $\mathbb{A}$. An Arrow Labelled Transition System (ALTS) is a labelled transition system $(\Gamma, T, \mathbb{A}, \to)$ where $\mathbb{A}$ is a category, Γ is the set of configurations, and $T \subset \Gamma$ the set of terminal configurations. Unlike in conventional SOS, configurations consist only of (abstract) syntax and computed values (value added syntax). Two transitions may be adjacent iff the composition of the their respective labels is defined.

Category $\mathbb{A}$	$\lvert\mathbb{A}\rvert$	$\mathbb{A}$	$\mathbb{I} \subseteq \mathbb{A}$	$\cdot;\cdot$
$Discrete(Env)$	ρ	ρ	ρ	$\rho;\rho = \rho$
$Pairs(Store)$	s	$(s1, s2)$	(s, s)	$(s1, s);(s, s2) = (s1, s2)$
$Monoid(Act^*, conc, [])$	$single$	a^*	$[]$	$a_1^*;a_2^* = conc(a_1^*, a_2^*)$

Table 4.1: Basic Label Categories

For the definition of programming languages in MSOS the basic label categories that interest us here are listed in table 4.1. They model fundamental ways of processing information. For example environments in the $Discrete(Env)$ category are only allowed to be inspected. On the other hand, stores in category $Pairs(Store)$ can also be updated, and the steps contained in $Monoid(Act^*$-$, conc, [])$ do nothing but occur. The comparison to conventional SOS is instructive: taking labels in $Discrete(Env)$ gives rise to a relative conventional SOS relation $\rho \vdash e \to e'$. Taking labels in $Pairs(Store)$ corresponds to an unlabelled transition system in which configurations have the form (e, s). Finally, the actions forming $Monoid(Act^*, conc, [])$ are the labels of transition rules which are often used in process algebras (e.g. ACP [2]), with the empty sequence [] as the silent step τ.

The basic label categories listed in table 4.1 will be the components of our arbitrary label category except for the last one which is not used in the MSOS definition of AN2. The access of the different components is achieved with label transformers which allow the independent inspection and update of components of arbitrary label categories. Let $get : \mathbb{A} \times \mathit{Index} \rightarrow \mathit{Univ}$ and $set : \mathbb{A} \times \mathit{Index} \times \mathit{Univ} \rightarrow \mathbb{A}$ be operations that respectively retrieve and update components of some universe Univ of labels at index $i \in \mathit{Index}$ in arbitrary category $\mathbb{A}$. So, $get(\alpha, i)$ retrieves the category component $u \in \mathit{Univ}$ at index $i \in \mathit{Index}$ and $set(\alpha, i, u)$ returns a category with u as component at index i. For the definition of these transformers we refer to [31]. It is sufficient to see the arbitrary label as a total map $\mathbb{A} : \mathit{Index} \rightarrow \mathit{Univ}$ which itself obeys the laws of a category.

In this section we will only use three label components. These components consist of the categories $\mathit{Discrete}(\mathit{Env})$ at index *data*, $\mathit{Pairs}(\mathit{Store})$ at index *storage*, and finally $\mathit{Pairs}(\mathit{Cells})$ at index *cells*. We let Cells range over sets of cells, indicating the cells that ever have been allocated (but possibly deallocated). Furthermore we define for the latter two categories: $get_{pre}(\alpha, i) = s$ and $set_{post}(\alpha, i, s'') = set(\alpha, i, (s, s''))$, if $get(\alpha, i) = (s, s')$. To illustrate the implementation of MSOS transition rules in Asf+Sdf We will use a fragment of the MSOS specification of Kernel AN2. This fragment is displayed below[1]:

$$\textbf{\textit{provide}}\ d \xrightarrow{\iota} \mathit{normal}\ d \tag{4.1}$$

$$\frac{A_1 \xrightarrow{\alpha} A_1'}{A_1\ \textbf{\textit{then}}\ A_2 \xrightarrow{\alpha} A_1'\ \textbf{\textit{then}}\ A_2} \tag{4.2}$$

$$\frac{\alpha' = set(\alpha, data, d_1) \wedge A_2 \xrightarrow{\alpha'} A_2'}{\mathit{normal}\ d_1\ \textbf{\textit{then}}\ A_2 \xrightarrow{\alpha} \mathit{normal}\ d_1\ \textbf{\textit{then}}\ A_2'} \tag{4.3}$$

$$\mathit{normal}\ d_1\ \textbf{\textit{then}}\ t_2 \xrightarrow{\iota} t_2 \tag{4.4}$$

$$\frac{\begin{array}{c} get_{pre}(\iota, storage) = s\ \wedge\ get_{pre}(\iota, cells) = cs\ \wedge \\ get(\iota, data) = sv\ \wedge\ \neg(c \in cs)\ \wedge \\ \alpha = set_{post}(set_{post}(\iota, storage, s[sv/c]), cells, (cs \cup c)) \end{array}}{\textbf{\textit{create}} \xrightarrow{\alpha} \mathit{normal}\ c} \tag{4.5}$$

4.2.2 Functional Implementation

Instead of asserting under what 'label circumstances' two configurations are in the transition relation, we are interested in computing the target object (state) of a label category and the target configuration *given a source configuration and a source object* (its input state). It is assumed that s is a ground term and that from this a ground object $o \in |\mathbb{A}|$ will result (or non termination). We could have used the "list-of-successes" approach to non determinism as advocated in [20]. Failure is then represented as the empty list of results. There are two problems with this approach. First, in AN2 failure is hardwired in the language as the terminal configuration "failed". Secondly, to be useful, computing a list of successes requires a lazy language, and Asf+Sdf is not such a language. The only non-deterministic construct in AN2 is the $\textbf{\textit{select}}(A_1\ \textbf{\textit{or}}\ \ldots\ \textbf{\textit{or}}\ A_n)$ action. It performs any of the A_i first and, on failure of the selected action, performs

[1]Our syntax differs slightly from the one used in [32] from which these definitions are taken.

```
[sos1]  s |- provide d = <normal d, s>

[sos2]  s@data = d
        ===
        s |- copy = <normal d, s>

[sos3]  s |- A1 = <A1', s'>
        ===
        s |- A1 then A2 = <A1' then A2, s'>

[sos4]  s' = s[d1/data], s' |- A2 = <A2', s''>
        ===
        s |- normal d1 then A2 = <normal d1 then A2', s / s''>

[sos5]  s |- normal d1 then t2 = <t2, s>

[sos60] s@storage = st, s@cells = cs,
        s@data = sv, storable(sv) = true,
        c = new-cell(cs), cs' = insert(c, cs),
        s' = s[st[sv/c]/storage][cs'/cells]
        ===
        s |- create = <normal c, s'>
```

Figure 4.2: Fragment of Kernel AN2 specification in ASF+SDF

any of the remaining choices. If the selected A_i does not fail no other choices are performed[2]. To implement the semantics in ASF+SDF (instead of *defining* it) we will use the function $\cdot \vdash \cdot : |\mathbb{A}| \times \mathit{Action} \to \mathit{Action} \times |\mathbb{A}|$. This function should obey the following constraint:

Constraint 1 (Soundness) *For each computed state s' from an input state s there should be an arbitrary label category α for which the following holds:*

$$s \vdash A = \langle A', s' \rangle \;\Rightarrow\; A \xrightarrow{\alpha} A' \;\wedge\; \mathit{pre}(\alpha) = s \;\wedge\; \mathit{post}(\alpha) = s'$$

The implementation of the MSOS rules listed earlier is displayed in Figure 4.2. Note that the equations that return the incoming state s correspond to rules of the form $c \xrightarrow{\iota} c'$. The auxiliary operator $\cdot/\cdot : |\mathbb{A}| \times |\mathbb{A}| \to |\mathbb{A}|$ (in equation `[sos4]`) is used to address the problem of enforcing that one transition rule may change one component of a structure without changing, or even knowing about, the rest? For the two label categories that we used, this operator is defined on $|\mathbb{A}|$ by $\rho_1/\rho_2 = \rho_1$ and $s_1/s_2 = s_2$ (where $\rho \in \mathit{Env}$ and $s \in \mathit{Store}$). The idea of this operator is to let changeable states change and fixed states remain the same. An example may clarify why this operator is needed. Consider the action *normal* 1 ***then*** *copy* and let $\varepsilon = []$ (the empty map). Then we get:

$$\varepsilon \vdash \;\mathit{normal}\ 1\ \textbf{\textit{then}}\ \mathit{copy} = \langle \mathit{normal}\ 1\ \textbf{\textit{then}}\ \ \mathit{normal}\ 1, [] \rangle$$

In absence of this operator the result of this step would have been:

$$\langle \mathit{normal}\ 1\ \textbf{\textit{then}}\ \ \mathit{normal}\ 1, [][1/\mathit{data}] \rangle$$

which is a violation of constraint 1.

[2]This scheme is also known as “don’t care choice”.

4.2.3 Modularity of `evalan`

Modular SOS has been designed to achieve better modularity for SOS descriptions of programming languages. That is, transition rules for different facets of a language can be defined independently, without knowing of each other. This has an important advantage, especially when a language is in design stadium. Modules can be changed according to need, without having to reformulate all the other rules. But there are other modularity issues, more related to software engineering aspects. These issues have more relevancy for implementation than specification, but implementation *is* what we are doing here. One of the drawbacks of composing modules in exactly the same way as the MSOS specification in CASL is that (abstract) syntax is defined alongside the semantical rules. This entanglement makes it impossible to use the syntax for different purposes without including all semantical information with it. So, the actual implementation of `evalan` is divided over two *threads* of modules in such a way that the syntactic modules are available without the equations that define the meaning of the constructs defined therein.

Another problem that had to be addressed is to make the addition of arbitrary data types, operations and predicates as natural as possible. The semantics of AN-K does not contain any axioms for data notation (except for bindings and tuples). The Data Notation is a parameter of Action Notation. Since passing functions around by name is restricted to prefix functions in ASF+SDF, this feature had to be mimicked. The syntax of data operations and predicates is defined in SDF as:

```
"_" DataOpInfix "_"    -> DataOp
DataOpPrefix "_"       -> DataOp
"_" DataPredInfix "_"  -> DataPred
DataPredPrefix "_"     -> DataPred
```

The `*Infix` and `*Prefix` sorts are left unspecified. In AN data operations are enacted by the primitive action ***give*** o and predicates are asserted by ***check*** q. The equations that implement their meaning use two auxiliary functions:

```
result DataOp Data  -> Data
holds DataPred Data -> Boolean
```

Although these functions seem to be defined on all values included in sort Data, they have a default meaning indicating that this is not so:

```
[default-result]  result o d = nothing
[default-holds]   holds q d = false
```

These equations fire when all others fail. $Nothing$ is the 'undefined datum'. Assume that we would like to have complex numbers added to our PICO^{-} language definition from chapter 2. The module defining complex numbers is displayed in figure 4.3. The first rule defines a complex number as a pair of Reals. The sort Complex is then injected (not *included*, see below) into the sort of individual data values. Finally, two infix data operations are declared and a sortname. In the equations section of this module the meaning of the data operations is expressed using the aforementioned result function. The first equation extends the meaning of the standard ascription data operation **the_**. The matching constraint induced by variable C, ensures that these equations do not fire when

```
module Complex
imports Real Data-Interface
exports sorts Complex
  context-free syntax
    Real # Real  -> Complex
    Complex      -> Datum
    "c+"         -> DataOpInfix
    "c*"         -> DataOpInfix
    "complex"    -> SortName
variables "C"[0-9\']* -> Complex [abcd]* -> Real
equations
  [1]  result the complex_ C = C
  [2]  result _c+_ (<a,b>,<c,d>) = <a+c,b+d>
  [3]  result _c*_ (<a,b>,<c,d>) = <a*c-b*d,a*d+b*c>
  [4]  storable(C) = true
  [5]  bindable(C) = true
```

Figure 4.3: Adding complex numbers to `evalan`

called with data arguments other than complex numbers. Finally, the last two equations express that complex numbers are both storable and bindable. These two predicates, *storable* and *bindable*, are needed because ASF+SDF is not a subsorted formalism. The use of injections ("invisible functions") can only remedy this in a limited way. The problem can best be explained using an example. Let's pretend we can use an injection to express a sort inclusion. The following productions are relevant here:

```
Integer  -> Storable
Integer  -> Bindable
Bindable -> Datum
Storable -> Datum
Datum    -> Data
```

This is perfectly valid in formalisms such as CASL and Maude, and it is valid in SDF but one gets unexpected results if these rules are interpreted as declaring a sort inclusion. Intuitively it is no more than reasonable to consider integers bindable as well as storable, and it is required that both storables and bindables are "included" in the sort Datum. Consider the action term *provide* 1 ***then*** *create*. The parser derived from the syntax specification of Action Notation will report ambiguities when it tries to parse the example action. One can shuffle around these injections to resolve the ambiguities but none of the solutions are natural or satisfying. The problem is similar to the one encountered in single inheritance languages such as Java, which introduces interfaces to solve the problem. Our solution is similar to the interface approach as well, since asserting *storable* for a sort S will also apply to the sorts S' injected into it, just as classes in Java inherit the implemented interfaces from their super classes.

4.3 Term Rewriting Actions in C

4.3.1 Preliminaries

As mentioned in chapter 3 the implementation of a term rewriting system for actions in C was greatly alleviated by our prototype in ASF+SDF. Before we dive into the implementation aspects of `acr`, we list here the crucial differences between `acr` and `evalan`:

- `acr` takes abstract syntax trees (ASTs) as input, whereas `evalan` operates on concrete syntax trees. Both kinds of trees are represented internally by Annotated Terms (ATerms) [7]. The abstract syntax trees are obtained by "imploding" concrete syntax trees, which is a language independent transformation that removes layout, injections, production annotation and so on. The obvious advantage is that parse trees obtained from action semantic descriptions of programming languages in ASF+SDF can be fed into `acr` with only one intermediate step, *while retaining maximal sharing.*
- In `acr` imperative actions are really imperative, and storage is really global, i.e. accessible to all agents. This means that storage (or any other object of category *Pairs*) is not passed as an argument to the functions that implement the meaning of each action sort.
- The supported Data Notation is implemented using ATerms as well. It is in fact the same library of data operations that is used in the AIL approach (see Chapter 6). There is however one difference: Action datums (and reflection functions) operate on ASTs in `acr` whereas the AIL tools use a bytecode representation.

4.3.2 Design of `acr`

To process imploded parse trees representing arbitrary actions, a method is needed to decompose and construct action terms. One way to do this is to explicitly code the structure of the AST into the functions that implement the semantics of a particular action. However, if the structure of Action Notation changes, many functions may have to be modified to comply to the new abstract syntax. Therefore, we employed a tool developed at CWI, called `apigen` [22] which generates a C interface from an SDF grammar definition based on the ATerm library. Thus, the grammar of AN as part of `evalan` is efficiently reused for generating the syntactic interface of `acr`. The fact that ASTs are internally represented by ATerms is now (almost) completely hidden from the implementation of `acr`.

The semantical functions for the different sorts of actions are hierarchically structured according to the sort structure of the grammar of AN. So the top level function for one execution step accepts the most general sort as a parameter, an Action. It is then decided whether to deal with an action constant, an action combinator or a special action. This proceeds until a concrete action constructor matches the argument and the appropriate action is taken. These action handlers all have the following form:

static ANK_Action ACR_handle⟨*Sort*⟩(ANK_Action action,
  AN_Data data, AN_Data bindings);

An action handler thus accepts three parameters: the action term itself, the given data and the current bindings. The function returns a reduced action. The fact that no data or bindings are returned effectively captures the desired semantics of a *Discrete* category, which ensures that data and bindings are local and must not be changed (they are read only values). There are no other parameters, since they are supposed to be objects of a *Pairs* category; these objects are global and may change. To achieve this in a purely functional style, the action handlers should have returned a tuple of a reduct and a set of changeable states (i.e. storage, cells, schedule etc.). It should be noted that these action handlers model a one-step execution of an action, so it should be relatively easy to add tracing and stepping facilities to `acr` to alleviate debugging of executing actions.

```
ANK_Action reduce(ANK_Action action, AN_Data data, AN_Data
bindings) {
  register ACRAgent agent;
  register ANK_Action reduct;
  ACR_init_schedule(&schedule, ACR_main_agent());
 loop: {
    agent = ACR_schedule_get_current(schedule);
    if (ACR_is_main(agent)) {
      reduct = ACR_handleAction(action,data,bindings);
      action = reduct;
    }
    else {
      reduct = ACR_agent_get_action(agent);
      reduct = ACR_handleAction(reduct,data,bindings);
      ACR_agent_set_action(&agent,reduct);
      ACR_schedule_update_current(&schedule, agent);
    }
    if (ANK_isValidTerminated((ANK_Terminated)reduct))
      ACR_schedule_remove(&schedule,agent);
    if (ACR_schedule_is_empty(schedule))
      return action;
    ACR_schedule_rotate(&schedule);
    goto loop;
  }
}
```

Figure 4.4: Top level reduce algorithm with scheduling

The full evaluation of an action is enacted by the top *reduce* function which also copes with scheduling of agents. The scheduling policy of *reduce* slightly differs from the one in the Modular SOS definition of Kernel AN2. The algorithm, as displayed in Figure 4.4, terminates when there are no executing agents left. The MSOS specification, on the other hand, terminates when agent main (the input action) has terminated. This is in contrast to the informal introduction to AN2 where the authors state that "Action performance by agents is *fair*: the performance of every action that has not terminated eventually proceeds" [24]. We chose the *fairness* approach since it seems the most reasonable and is applied in many other programming languages as well (e.g. Java).

4.3.3 Extension of `acr`

In this section we present a solution to making `acr` extensible with user defined primitive actions. The current implementation of `acr` is only an interpreter of predefined actions and data operations, so this section can be seen as future work. The extensibility plan presented here would enable `acr` to be used in real software environments in a generic and portable way. The idea is to link `acr` to the Scheme interpreter Guile [19] which is available as a library. This embedded interpreter can be used to evaluate arbitrary functions defined in Scheme. A user can thus define functions in Scheme of a certain signature which can be called on request during the rewriting of an action by `acr`.

Consider one would like to add a **write** primitive to PICO$^-$ which prints a message to the console. While such a primitive is almost too simple to be absent, extending `acr` with it would mean adding an action handler to `acr` and recompiling the interpreter. This is clearly not a desirable state of affairs. How can this be done in a flexible and user driven way? Let us sketch a trajectory of steps to accomplish the goal of adding **write** to AN-K using the embedded Scheme interpreter. We will leave the details of extending the PICO$^-$ language aside, since it is trivial. First we have to define a new primitive action *write* which takes a datum and gives the empty tuple. In SDF this would look like:

```
"write" -> ActionConst {cons("user-action:write")}
  %% [taking datum, giving ()]
```

The crucial thing here is the cons(tructor) annotation "user-action:write". These *cons* attributes are used by the parse tree implosion tool to give names to production rules. Imploding the parse tree of "write" will result in an AST `const("user-action:write")`. The function symbol `const` is a consequence of the SDF production: `ActionConst -> Action {cons("const")}`. Since the abstract syntax for *write* is not part of the standard Action Notation, we have to add one conditional to `acr` that detects the use of the pattern `user-action:*`. If such a pattern is used the second half of the matching term is used to call the appropriate Scheme function. So `acr` assumes that the user has provided for a scheme function:

```
(define (write data bindings) ...)
```

This kind of user defined functions in Scheme can be loaded dynamically by `acr`. After `acr` data is converted to scheme data, the function is called, and the result is converted back to valid `acr` data.

Chapter 5

Compilation of Actions

5.1 Introduction

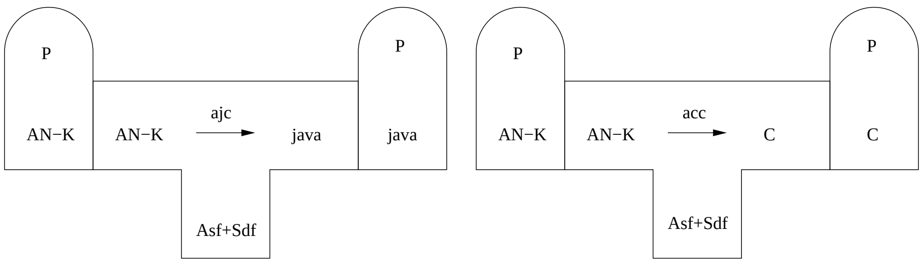


Figure 5.1: Compilation of AN-K to Java resp. C

In this section we want to discuss the architecture of `acc` and `ajc`. Our presentation, however, is based on the compilation to C. Again we display the T-diagrams from chapter 3 as a reference in Figure 5.1. The compilers accept a parsetree over Kernel AN2 syntax and respectively produce Java and C code. The compilers are specified entirely in ASF+SDF in a layered fashion according to the structure of the source language (i.e. AN2). The layered compilation of actions has a number of advantages. First, an action term is compilable using only the lowest level of representation. Second, the addition of direct compilation of Full AN2 constructs is very easy. Thus, it is possible to add optimizations without having to change the compiler for Kernel AN. First we will dive into the specification of the action to C compiler (`acc`). Then we will address in what sense the presented compilation scheme is generic. The final two sections contain descriptions of the runtime environments of `acc` and `ajc`.

5.2 Layered Compilation

5.2.1 `acc` as an Example

The compilation of actions to C involves five functions which are declared as follows:

```
fa-to-identifier(Action,Environment) -> ID
fa-to-expression(Action,Expression,Expression,Environment) ->
  Expression
fa-to-default-expression(Action,Environment) -> Expression
fa-to-decls(Action,Environment) -> ExternalDefinitions
fa-to-function(Action,FunctionBody,Environment) ->
  FunctionDefinition
```

The `fa-` prefix indicates that these functions are meant to be applied to Full AN2 constructs. The functions can be seen as a family of functions parameterized by the layer they are defined for;—in this case the level of Full AN2 (`fa`). The sort *Environment* is used to pass additional data used for generation of identifiers etc. These functions are specialized for Kernel Actions in a different module and prefixed with `ka-`. Let us refer to these function as the "kernel functions". In that module, there are five *default* equations for the Full AN2 functions. These default equations all obey the following pattern:

```
[default] fa-to⟨Sort⟩(A,env) = ka-to⟨Sort⟩(A,env)
```

The kernel functions have default equations as well.

```
[default-k1] n = env.pos
             ===
             ka-to-identifier(A, env) = make-id(A, n)

[default-k2] ka-to-expression(A,exp1,exp2,env) =
              pa-to-expression(A,exp1,exp2,env)

[default-k3] ka-to-decls(A,env) =
```

The default identifier for an action is an identifier (e.g. "action") together with the position in the action tree. The position of an action is kept track of in the generic environment. The function `make-id` constructs a valid C identifier. If the default equation for `ka-to-expression` fires, we are dealing with a primitive action, so the appropriate function is called. Default declarations for an action A are assumed to be empty, since this equation only fires for primitive actions and they are predefined by the runtime library of `acc`. The fact that only primitive actions make the last two equations fire, is a consequence of the equations that will follow: they are only defined on combined actions.

The translation of actions to (default) expressions can be specified without default equations.

```
[kde]   ka-to-default-expression(A,env) =
         fa-to-expression(A,data-parameter(),
                          bindings-parameter(),env)

[ke1]   ka-to-expression(A1 Ai A2,exp1,exp2,env) =
         fa-to-identifier(A1 Ai A2,env)(exp1,exp2)

[ke2]   ka-to-expression(Ap A,exp1,exp2,env) =
         fa-to-identifier(A Ap,env)(exp1,exp2)
```

The first equation compiles any action (including primitive actions) to a so called default expression. This is simply a convenience function which maps an action to a C expression with the default arguments for given data and bindings. These identifiers are produced by the function `data-parameter` and `bindings-parameter`. If these arguments are supplied the second two equations apply. These equations are only defined for combined actions and construct a function call using the the Full AN2 function `fa-to-identifier`. To clarify why these last two equations are only defined for combined actions, we can best take an example action and follow the process of compiling that action. Take A to be the action *provide d*. This action does not take any input. Evaluation of `a-to-default-expression(A,env)` leads to the following trace:

```
fa-to-default-expression(provide d,env)        ⇒
 ka-to-default-expression(provide d,env)       ⇒
 fa-to-expression(provide d,data,bindings,env) ⇒
 ka-to-expression(provide d,data,bindings,env) ⇒
 pa-to-expression(provide d,data,bindings,env) ⇒
 an-data-to-c-data(d)
```

The function `pa-to-expression` maps actions of the form *provide d* to d with d converted to an appropriate C expression. The point is, that if the regular equations of `ka-to-expression` were defined on primitive actions as well, A would have been compiled to a function call which is sheerly redundant. The same applies to the primitive action *copy* for which `pa-to-expression` returns the identifier "data" (which resulted from evaluating `data-parameter`).

There is one more generic function which has not yet been described. This function takes an action, a list of statements and an environment and then produces a function definition.

```
[kf]    ka-to-function(A, decls stats, env) =
static data-type() fa-to-identifier(A,env)(
             data-type() data-parameter(),
             bindings-type() bindings-parameter())
{
   decls
   stats
}
```

A C signature is produced using the same identifier function as is used in the equations which map actions to expression to guarantee that actions at some position in the tree get the same identifier both in declaration and invocation. The discussion of these preliminaries may seem very abstract for the moment but they become perfectly clear if we turn our attention to the compilation of combined actions to C function declarations. All the functions described this far are used in the following equation which compiles actions of the form A_1 ***then*** A_2:

```
[kd1]   env1 env2 = split-environment(env)
        ===
        ka-to-decls(A1 then A2, env) =
fa-to-decls(A1, env1)
  fa-to-function(A1 then A2,
        data-type() d1 = fa-to-default-expression(A1,env1);
        data-type() d2 = fa-to-expression(A1,d1,
                         bindings-parameter(),env2);
        return d2;,
      env)
    fa-to-decls(A2, env2)
```

The condition of this equation splits the current environment into two new environments. In this environment there is, *at this level*, only one component: a position integer. When an environment is split, the new environments contain fresh positions. This way unique identifiers for any action are guaranteed. The declarations for A_1 and A_2 are produced using `fa-to-decls` before and after the actual function definition of A_1 ***then*** A_2. The body of this function contains three statements. The first statement computes A_1 using `fa-to-default-expression`. This means that this action gets data and bindings from the arguments of the defined function (see `fa-to-function`). In the second statement the result of A_1 is input to A_2, thus capturing the semantics of ***then***. Finally, the result of A_2 is returned. Note that all functions in the righthand side are Full AN2 functions which in the current setting call the kernel functions (via the default equations presented above).

Overriding Default Behaviour

To prevent the use of reflection in the case of ***unfolding*** A actions, we will now override the default (i.e. kernel) behaviour of the compiler. This is accomplished by providing (non-default) equations for the `fa-` family of functions. These functions until now have only default equations which evaluate to the `ka-` family semantics.

First we have to extend the environment which is used to pass additional information down the action tree during compilation. This is done in a safe way such that the kernel functions do not know about the extension, but pass it down anyway. The environment is extended with a field referenced by index “unf” which contains a tuple of an action and boolean. The action is the action within ***unfolding***,—the boolean indicates whether this action is tail-recursive. By tail-recursive we mean in this context: *the result of unfold is not used within* ***unfolding***.

Now we can define the compilation of ***unfolding*** A. The first equation deals with the non-tail-recursive case:

```
  []      is-unfolding(A) = true, get-unfolding(A) = A',
          is-tail-recursive(A) = false,
          n = env.pos, n' = n + 2,
          env1 = env.unf := <A', n', false>,
          env2 = env1.pos := n'
          ===
          fa-to-decls(A, env) =
            fa-to-decls(A', env2)
            fa-to-function(A,
             return fa-to-default-expression(A', env2);,env)
```

First the unfolding action is retrieved in variable A' and it is checked that A' is not tail recursive. Then A' is saved in the environment together with a fresh position n'. Finally the result consists of the declarations for A' and a declaration for ***unfolding*** A'. This last declaration consists of an invocation of A'. Note that we do not have to change the function which deals with the compilation to an invocation of the unfolding action because both declarations and expressions are compiled using the identifier generated from the position in the tree.

For this case, the compilation of *unfold* also needs special attention:

```
[]      is-unfold(A) = true,
        <A',n,false> = env.unf,
        env' = env.pos := n
        ===
        fa-to-expression(A, exp1, exp2, env) =
           fa-to-expression(A', exp1, exp2, env')
```

If A is the action constant *unfold* we retrieve the corresponding action from the environment (at the same time checking that it is not tail-recursive). Since we need the expression corresponding to the position of A', the expression is produced using the previously saved position. Note that the result of this equation is equal to the result of `fa-to-default-expression` in the equation for ***unfolding*** A, modulo the data and bindings parameters. For *unfold* no declarations have to be generated, so an additional equation is needed which produces empty declarations for *unfold*.

If an unfolding action turns out to be tail-recursive, slightly more machinery is needed. We will therefore present no equations, but describe the translation informally. The idea can be explained as follows. Since the result of *unfold* will not be used, *unfold* can jump to the start of the loop. Note that by "jump" we mean a non-local jump, since all actions are compiled to functions. The action ***unfolding*** A at term position p is compiled to:

```
ACCData unfolding_p(ACCData data, ACCData bindings) {
  loop: if (!ACC_unfolding(&data, &bindings))
          return [[A_p0]](data, bindings);
        goto loop:
}
```

The loop terminates when the return statement really returns something (i.e. after an iteration without *unfold*). The compilation of ***unfolding*** has much resemblance to the compilation of exception handling and non-deterministic choice. `ACC_unfolding` returns 0 when setting a choice point. Then A_{p0} is executed. *Unfold* will *fail* and jump to the corresponding choice point. *Unfold* is compiled to:

```
return ACC_unfold(data,bindings);
```

The procedure `ACC_unfold` saves the arguments on a stack and *fails*, thus jumping to the last `ACC_unfolding`. At that point, the `data` and `bindings` are updated with the top of the stack, and 1 is returned. Then the loop is reenacted.

Earlier we remarked that this way of coping with tail-recursive *unfoldings* uses techniques similar to exception handling. To reenforce this point, we will now describe the compilation of ***unfolding*** A to Java for the tail-recursive case. The Java-compiled version of ***unfolding*** A actually uses standard exception handling mechanisms.

```
private Data unfolding_p(Data data, Data bindings) {
  while (true) {
    try { return [[A_p0]](data, bindings); }
    catch (Unfold unf) {
      data = unf.getData();
      bindings = unf.getBindings();
    }
  }
}
```

Now *unfold* boils down to just throwing a new *Unfold* exception:

```
throw new Unfold(data, bindings);
```

5.2.2 Some Notes about Genericity

This subsection discusses the generalization of the compilation scheme. The layered compilation scheme presented above evolved from the need to separate the compilation of Kernel AN2 and Full AN2 and to allow further optimizations to be added without having to change the complete compiler. In fact the compilation scheme can be viewed in an abstract way and be reused to target more languages. In fact, we have done this for the compilation of AN2 to AIL (Chapter 6): it is specified in exactly the same way as the compilation to Java and C.

To abstract away from source and target language let us consider an example of an unspecified layered function. Let the following ASF+SDF module be a template for declaring a layered family of functions:

```
module Function
imports basic/Integers
exports
context-free syntax
  f(X̄) -> Y
  f(Integer,X̄) -> Y
  ⊤ -> Integer
  ⊥ -> Integer
equations
  [default] f(n,x̄) = f(n − 1, x̄) when n != ⊥
  [actual]  f(x̄) = f(⊤, x̄)
  [bottom]  ⊥ = 0
  [top]     ⊤ = ⟨highest level⟩
```

The module *Function* declares one function f but it could just as well declare many different functions that should be "layered". Function f accepts a vector $\overline{X}$ of parameters which are left unspecified in this example; we call this function *the actual function.* The result sort is Y. Next the function f is overloaded with an additional argument of sort `Integer` which is used to designate a level[1]. Finally two tokens are declared which denote the top (highest) level and the bottom (lowest) level. To let the default versions of the overloaded version of f

[1]One could use terms that are more informative then just integers by defining a function $pred(X) \to X$ for a sort X which contains names of levels.

reduce to the same functions one level below, the first default equation is given. Note that the `default` tag is crucial here, since the equation should only fire if there are no normal equations defining the overloaded f for level n (defined elsewhere). The meaning of the actual function f is defined to be equal to the meaning of the overloaded function evaluated for the top level; this is stated in the second equation. The meaning of $\top$ and $\bot$ is specified in the last two equations. If the function f is to be divided over l levels, $\top$ should evaluate to $l-1$. We can then produce modules M_i ($\forall i : 0 \leq i < l$) such that:

- M_i imports M_{i-1}
- M_0 imports *Function*
- each M_i provides equations for $f(i, \overline{x})$
- all equations in M_i ($0 < i < l$) are normal (non-default); equations in M_0 should be exhaustive (i.e. may be default)
- any recursion on f is performed using the *actual* function f; that is, the overloaded f is never directly evaluated.

There is one aspect of dividing a function over a number of levels that we have only mentioned yet. This is the extension of the signature of f in higher levels. In our compilers one of the arguments in $\overline{X}$ is an environment which contains additional information used to compile actions. In essence, this environment contains (extra) arguments to f. For the compilation of Full AN2 constructs, this environment is extended; that is, f gets more input. However, by hiding these arguments in one (environment) argument, we can extend the signature of f without having to change the specifications of lower levels (i.c. the kernel compiler).

The advantages of layering a function over a number of levels are that every level can be used in isolation, and, if a level is added, we only need to produce M_l and increase the value for $\top$.

5.3 ACC Internals

5.3.1 Overview

The runtime system used by `acc`-compiled actions consists of a Data Notation, non-local jump primitives and a reflection interface. The reflection interface is discussed in more detail in the next subsection.

The Data Notation is implemented using the ATerm Library [7, 6]. This implementation of a Data Notation is the same as is used by `acr` and `avm` (see Chapter 6). The main advantage of using the ATerm library lies in the fact that ATerms are garbage collected. This way no static liveness analysis for variables is needed. Secondly, since AN2 is mostly a functional language, the copy-on-write semantics of ATerm modification fits the semantics of Data Operations very neatly.

Non-local jumps are needed to implement exception handling, non-deterministic choice and tail-recursive unfoldings. All three uses are based on the same choice point library [26]. The runtime library defines some additional wrap-around functions to distinguish exceptions, failures and unfolds, and to be able to pass data with thrown exceptions and unfolds.

The interacting facet is not supported by `acc`. This is our one exception to the requirement of supporting the full kernel of AN2. One would think that executing agents could have been mimicked by using threads as provided by the operating system. The reason that we have done not so is a consequence of the use of ATerms: the garbage collection algorithm of the ATerm library cannot cope with threads, since it uses the runtime stack to detect whether some term is referenced or not.

5.3.2 Reflection: Lightning

Constructive reflection in a language like C seems to be a great challenge. It would have been if there would not have existed a platform independent just-in-time assembler: GNU Lightning [4]. Lightning consists of a number of header files containing macros to define ordinary C functions at runtime. These macros are abstract in the sense that their interface is platform independent. Using the macros, however, results in real assembly for Sparc, i386 and PPC platforms. An example may illustrate how Lightning can be used to achieve AN2 reflection at the level of C. The following listing describes how the data operation *give _then_* is performed.

```
ACCFunctor ACC_jit_then(ACCFunctor a1, ACCFunctor a2) {
  char *start, *end; int data_offset, bindings_offset;
  ACCFunctor f;
  f = (ACCFunctor)(jit_set_ip(cur_ip).pptr);
  start = jit_get_ip().ptr;
  jit_prolog(2);
  data_offset = jit_arg_p();
  bindings_offset = jit_arg_p();
  jit_getarg_p(JIT_V0, data_offset); /* v0 = data */
  jit_getarg_p(JIT_V1, bindings_offset); /* v1 = bindings */
  jit_prepare(2);
  jit_pusharg_p(JIT_V1);
  jit_pusharg_p(JIT_V0);
  jit_finish(a1);
  jit_retval(JIT_V2); /* v2 = a1(data,bindings) */
  jit_prepare(2);
  jit_pusharg_p(JIT_V1);
  jit_pusharg_p(JIT_V2);
  jit_finish(a2);
  jit_retval(JIT_RET); /* ret = a2(v2,bindings) */
  jit_ret();
  cur_ip = jit_get_ip().ptr;
  jit_flush_code(start,cur_ip);
  return f;
}
```

Recall that the type ACCFunctor is a functiontype that accepts data and bindings as input and returns data as a result. This kind of functions are input to

`ACC_jit_then`. All identifiers starting with "jit" (both lowercase and uppercase) are part of Lightning. The first statement sets the internal instruction pointer to the memory region pointed to by `cur_ip`. The instruction pointer is cast to a function pointer returning a pointer and is then stored in `f`. The start of the function is saved in `start`. Then the real just-in-time compilation starts. A standard prologue for a function accepting two arguments is produced by `jit_prolog(2)`. The offsets of the arguments are saved in two local variables. At line 9 and 10 these offsets are used to store the arguments in general purpose registers V0 and V1. Now that the arguments `data` and `bindings` are available we have to call the first argument `a1` with this input. Lines 11–14 perform this function call. At line 15 the result is stored in register V2. The next function call (lines 16–19) gets V2 as data input instead of V0, thus capturing the functional composition semantics of ***then***. Then the result of `a2` is put in the return register RET so that the result of performing `a2` is also the result of the composition. Finally, the instruction buffer is flushed, making the function available in `f`. The resulting function `f` can be invoked just like any other C function.

Some Issues

Currently there is no garbage collection of jitted actions. Actions as data are embedded in ATermBlobs. This should enable the automatic garbage collection of reflective actions but the problem is that all just-in-time compilations are performed in one region of memory (which should never be destroyed). A possible solution is to allocate a separate region of memory and copy the jitted function into it. This way overflow of the instruction buffer used by Lightning is prevented and functions can be garbage collected[2].

Another problem is a consequence of AN2 itself: constructive reflection is used abundantly in mapping Full AN2 actions to Kernel AN2 actions. On the kernel level, even simple ***unfolding*** actions heavily use reflection. When ***unfolding*** actions are directly compiled this problem is solved. However, when, for example, a closure with a specific input should be enacted, reflection is used. This is a consequence of the semantics of *enact*: it states that the action is enacted with no input and no bindings. To provide input, the action should first be composed with an input providing action using, e.g., *give _then_*. Reflection at runtime thus results in a jitted function for each time a closure is enacted. The current implementation memoizes jitted functions to prevent that multiple jits of the same function do not occur, but this only prohibits just-in-time compilation when the input to the closure is the same. Further research is needed, considering how to directly compile closures and enactments to avoid this kind of runtime reflection (just as has been done for ***unfolding*** and *unfold*).

[2]This is a consequence of the use of ATermBlobs. When an ATermBlob is collected, the memory it contains is freed.

5.4 Java Runtime Architecture

5.4.1 Overview

An action is compiled to a function in C. In Java actions are compiled to private methods of a public class which extends the class `AbstractEnactable`,—a base class which implements the `Enactable` interface. Reflective actions (actions provided as data) are compiled to a private inner class extending `AbstractEnactable`. Exception handling and non-deterministic choice is implemented using the standard exception handling mechanisms of Java. Two special exception classes are defined: `Exceptional` and `Failure` which are respectively caught by the translation of A_1 ***exceptionally*** A_2 and A_1 ***otherwise*** A_2. For the sequential exceptional combinator ***and exceptionally*** the same trick is used as in `acc` and `avm` (see Chapter 6): nested `try...catch` ensure concatenation of two exceptional results.

Storage and schedule are modeled by two Singleton [18] classes which are globally accessible. The schedule class takes care of sending and receiving messages. The actual parallel execution of agents (containing references to `Actions`) is implemented using the standard `Thread`/`Runnable` facility offered by the Java runtime. The classes `Store` and `Schedule` respectively implement the interfaces `Storing` and `Scheduling` to anticipate future changes. These interfaces have the following signatures:

```
public interface Storing {
 public Cell create(Storable storable);
 public Empty destroy(Cell cell) throws Exceptional;
 public Empty update(Cell cell, Storable o)
          throws Exceptional;
 public Storable inspect(Cell cell) throws Exceptional;
}
```

From this interface one can see that all actions pertaining to the imperative facet are all located in the same class. This propagates the orthogonality of the different facets of AN2 to the level of the Java implementation. Following this path of separation of concerns, the same can be said of the `Scheduling` interface:

```
public interface Scheduling {
 public Empty send(Agent agent, Message message,
           MessageTag messageTag) throws Exceptional;
 public Message receive(MessageTag messageTag)
           throws Exceptional;
 public Agent activate(Action action);
 public Empty deactivate(Agent agent) throws Exceptional;
 public Agent currentAgent();
 public Int currentTime();
 public Int chooseNatural();
}
```

All identifiers in the interfaces starting with an uppercase letter are interfaces from the Data Notation that we have implemented in Java, which is the subject of the following subsection.

5.4.2 Data Notation

Java interfaces provide a nice way to enforce certain features of classes without having to fall back on inheritance. In this subsection we describe (a part of) the structure of the Data Notation in Java which heavily relies on interfaces. The Data Notation provides classes representing some standard data types, and actions themselves. The actual implementation of the classes is hidden via the well-known Abstract Factory pattern [18]. All data operations and predicates (functions) have now become methods in the classes they are defined for. To create data the factory is the starting point. Currently there is only one factory implementation based on the standard Java classes. The translation of a subsorted Data Notation is first presented in a generic style. The next subsection, which discusses reflection, serves as an example.

Every sort S is mapped to an interface which extends the interface of its (primary) super sort. If S has more super sorts their respective interfaces are also extended. These super sorts are referred to as *features*[3]. For all data operations and predicates defined for sort S a method is added.

```
public interface ⟨Sort⟩ extends ⟨SuperSort⟩, ⟨Features⟩ {
  public ⟨Sort⟩ ⟨Data Operation⟩;
  public boolean ⟨Data Predicate⟩;
  ...
}
```

Implementations for S extend the implementation of their super sort and implement interface defined for S. Implementations are always based on some value which is hidden from the outside world. The implementation of data operations should be strictly functional (no update of local state) and return an object which obeys interface S. For data operations and predicates to have access to the internal representation a protected method `get⟨Value⟩Value()` should be implemented. Generically, such an implementation is an instantiation of the following template:

```
class ⟨Sort⟩Impl extends ⟨SuperSort⟩ implements ⟨Sort⟩ {
  private ⟨Value⟩ value;
  protected ⟨Value⟩ get⟨Value⟩Value() { return value; };
  public ⟨Sort⟩Impl(Factory factory, ⟨Value⟩ value) {
    super(factory);
    this.value = value;
  }
  // Implementations of the interface
}
```

Implementation classes are not publicly available. The creation of objects is deferred to the Factory.

5.4.3 Reflection: Enactables

Informally, an action can be defined as something that can be *enacted*. Apart from the *enactable* character however, actions of different kinds are also sorts

[3]Recall the example: *Int* ∈ *Datum*, *Int* ∈ *Storable*, *Int* ∈ *Bindable*. The last two supersorts are presumed to be features.

of data. That is, they are part of an inheritance hierarchy in Java. Classes which implement the `Enactable` interface serve as *values* for `Actions`. Thus, an action is an object which *contains* an `Enactable`.

```
public interface Action extends Datum, Runnable  {
    public Data enact() throws Exceptional, Failed, Unfold;
    public Action then(Action action);
    // ... other data operations
}
```

Since actions are included in the sort of Datum, this interface extends the interface `Datum`. Furthermore, an action can serve as part of an executing agent so the class Runnable is also included. Actions can thus be used to instantiate threads (agents). The concrete implementation (`ActionImpl`) extends the concrete Datum implementation (`DatumImpl`) and implements the `Action` interface. It is declared as follows:

```
class ActionImpl extends DatumImpl implements Action {
  private Enactable value;
  protected Enactable enactableValue() { return value; }
  public Action then(Action action) {
    return new Then(factory,enactable,action.enactableValue());
  }
  ...
}
```

The reflection methods declared in the `Action` interface above, construct subclasses of `ActionImpl` with specific local `Enactables`. Take for example the concrete class Then:

```
class Then extends AbstractInfixCombinator {
 private class _Then implements Enactable {
  public Data enact(Data data, Bindings bindings)
   throws Exceptional, Failed, Unfold {
   return enactable2.enact(enactable1.enact(data,bindings),
                           bindings);
   }
  }
 public Then(PureFactory factory,Enactable e1,Enactable e2) {
  super(factory, e1, e2);
  enactable = new _Then();
 }
}
```

This class extends an abstract class `AbstractInfixCombinator` (a subclass of ActionImpl) which declares two protected `Enactable` fields. These model the respective lefthand and righthand argument of an infix action combinator. Class `Then` contains an inner class which implements the `Enactable` interface. This inner class can thus serve as a value for an action. The implementation of the method `enact` captures the semantics of the ***then*** combinator. Finally when a `Then` object is constructed the result is an `Action` object which can be enacted just like any other `Action` descendant.

5.4.4 Multi Threading

Although the multi threading features of AN2 are implemented using the standard thread facility of Java, it is completely hidden in the class `Schedule` (an implementation of the `Scheduling` interface). It is instructive to have a look at the implementation of *activate* and *deactivate*, both defined in `Schedule`. Activate is implemented as follows:

```
public synchronized Agent activate(Action action) {
  Agent agent = factory.makeAgent();
  Thread thread = new Thread(action);
  agents.put(agent, thread);
  threads.put(thread,agent);
  buffers.put(agent, new TaggedBuffers(factory));
  thread.start();
  return agent;
}
```

The `activate` method takes an action as an argument. As we have seen, `Actions` implement the `Runnable` interface. So `Actions` can be used to initialize new Java threads. The resulting thread is put in a table with a fresh agent (produced using the data factory) as key. To retrieve an agent corresponding to a specific thread, another table contains the pair in reverse. Together the `agents` and `threads` tables form a bijective map. Then a new entry in the `buffers` table is created containing a tagged buffers structure. The class `TaggedBuffers` consists of a hashtable, with `MessageTags` as keys and queues of `Messages` as values. Such a buffer is used for sending (enqueuing) and receiving (dequeuing) messages. Finally the thread is started and the agent is returned.

The action *deactivate* is used to destroy the thread corresponding to a given agent:

```
public synchronized Empty deactivate(Agent agent)
    throws Exceptional {
  if (agents.containsKey(agent)) {
      Thread thread = (Thread)(agents.get(agent));
      thread.interrupt();
      agents.remove(agent);
      thread.remove(thread);
      buffers.remove(agent);
      return factory.makeEmpty();
  }
  throw new Exceptional(factory.makeEmpty());
}
```

The argument agent is looked up in the `agents` table and if a thread is found, it is interrupted. The agent is removed from the `agents` and `buffers` tables and the empty tuple is returned. If the agent does not exist an exception is thrown.

Currently, no attempt is made to influence the scheduling policy of the Java Virtual Machine. It is however possible to let the compiler insert `Thread.yield()` statements at appropriate points in the code. A second possibility is to have these statements in the implementation of all primitive actions, however, they then cannot be disabled if one would like to. Finally the scheduling policy can be dealt with in a central way, by having one separate thread responsible for scheduling. This thread has maximum priority and then is able to assign time slices to other threads (the running agents).

5.4.5 Issues Concerning Java as a Target

One of the advantages of using Java as a target over C is that Java is (almost) statically typed. This is especially rewarding when one chooses to compile Java to native code using GCJ, the GNU Java Compiler, which sometimes achieves even better results than ordinary C. However, our Data Notation in Java uses multiple inheritance for interfaces and this is not supported by GCJ. Multiple inheritance is a problem in another aspect of the Data Notation as well. The straightforward translation of subsorted algebraic specifications to an object oriented language should benefit from multiple inheritance. Thus it seems valuable to assess other target languages that compare well with Java with respect to other features. One such language is *Sather* [39]. Sather is an Eiffel-like object oriented language which supports multiple subtyping and inheritance, efficient compilation to C and a very high level concurrency interface. Compiling actions to Sather should be as easy as compiling actions to Java , if not easier. Alas, this language is not very heavily maintained.

Another language which is interesting as a target for compiling actions is *Erlang* [16]. Erlang is developed by Ericsson, is heavily used and maintained and is available in the public domain. It can be characterised as a functional, declarative language with enhanced support for concurrent processes and communication. One problem however with Erlang is that memory (storage) is local to processes, while in Action Semantics storage is global and accessible by all threads.

Chapter 6

AIL Internals

6.1 Introduction

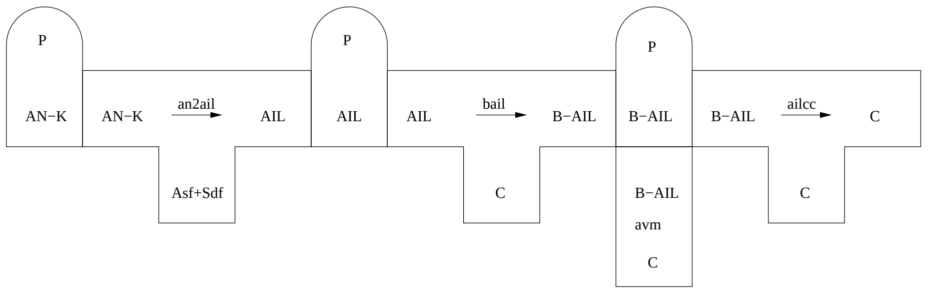


Figure 6.1: Compilation of Actions to AIL

One approach to executing actions is to transform an action to another intermediate format which is subsequently executed by an interpreter. In Chapter 3 we introduced a way to execute actions by compiling it to AIL. Until now, AIL stood for "Action Intermediate Language". In this chapter we show that AIL can also be understood as "Abstract Intermediate Language". During the design of the action virtual machine (`avm`), it became clear that the use of AIL was not restricted to the execution and compilation of actions. At the same time, pragmatic considerations motivated thorough reengineering of the way AIL was deployed. As the instruction set of AIL changed, the interpreter and byte code generator had to be both changed. The possibility to factor out most of the Action Semantic dependencies led to better software engineering practices and wider applicability. As such it can be seen as an exercise in software reuse. To acknowledge these considerations, this chapter is divided in two parts. The first part elaborates the generic parts of the AIL approach: we discuss the architecture behind AIL and present an example of how to use AIL. The second part describes how the AIL tools are used to produce an interpreter and compiler for AN. To connect this chapter to Chapter 3 we again display the T-Diagram in Figure 6.1. Recall that `bail` assembles symbolic AIL to binary AIL (B-AIL). The resulting object file can be executed by the virtual machine `avm` or be compiled to C with `ailcc`.

6.2 Generic Scaffolding for Virtual Machines

6.2.1 Preliminaries

The problem that AIL addresses can be stated as follows.

> *How to minimize programming efforts for prototyping virtual machine interpreters?*

This problem domain can be split in the following subdomains:

- A program P in language L should be mapped to an arbitrary sequence of instructions I_P which represents the semantics of P. I_P can be seen as an *assembly* language representation over A_L. The language A_L consists of generic syntax A for instruction sequences statically restricted by a signature of the set of valid instructions defined for L.
- The instruction sequence I_P should preferably be stored in some binary, non-symbolic form B_P.
- A virtual machine M_L implemented in some language should be able to execute B_P.

Taken together we need a compiler which *compiles* P into assembly I_P which is then *assembled* into B_P and finally *executed* by M_L. We assume that the compilation step is provided for by the user, specified in ASF+SDF. The assembly language A is AIL; its grammar is generically specified in SDF. AIL instructions should obey some rules. An instruction mnemonic is a sequence of alphanumeric characters. Each instruction can optionally have one parameter. This parameter can be a natural number (e.g. designating a register), a label (for branching instructions), a textual representation of an ATerm (e.g. constant values), or a complete sequence of instructions enclosed by curly braces (reflection). Finally, labels can be used to refer to positions in the sequence. Using ATerms as the primary datastructure is crucial since it allows serialization of constant data values into binary format.

Our solution to the problem of minimizing programming efforts for prototyping virtual machines consists of a number of generic components which are parametrized by definitions provided by the user. The first component we will discuss is a bytecode compiler (assembler) which translates AIL mnemonics to bytecode: `bail`. The ensuing binary objectcode can be subject to the following tools:

- Interpreter: a virtual machine partially generated by `genailapi`
- Compiler: a compiler to C partially generated `genailapi`
- Disassembler: completely generated by `genailapi`

The interpreter and compiler both depend on semantic information provided by the user and are therefore *partially* generated. The semantics of each instruction is the only thing the user will have to provide. This is done by implementing a number of C macros for which the signatures are generated. We will call these macros the *implementing macros*. The interpreter and compiler use the implementing macros to execute bytecode. The signature of each instruction is derived from AIL definitions, which are the subject of the next subsection.

6.2.2 AIL Definitions

AIL definitions are used to define instruction sets for a particular purpose. They consist of a list of instructions. Each instruction can optionally be followed by an argument pattern (one of “term”, “number”, “label”, “code”). An AIL definition is the syntactic reference for a subset of instructions which are parseable by the grammar of (generic) AIL. It contains sufficient information to build the binary structure of an AIL program and to decompose it. Therefore the Application Programming Interface (API) as generated by `genailapi` takes an AIL definition as input. From the signatures contained in the definition four entities are generated: a virtual machine architecture, a list of instruction prototypes, a compiler and a disassembler. Recall that the bytecode compiler `bail` is a separate tool, and is also parameterized by an AIL definition. Since the binary objectcode output by `bail` conforms to the AIL definition, it is guaranteed to be accepted by the virtual machine and other generated tools.

6.2.3 Bytecode Generation

The assembler `bail` takes an AIL program P and and AIL definition S as input. P is assumed to be a parsetree over the AIL grammar as produced by a user provided *L-to-AIL* mapping. The parsetree S is a term of sort AIL-Definition. Both input objects are thus parsetree representations over two grammars: AIL and AIL-Definition. To access these parsetrees in a parsetree format independent way in C, we used a tool developed at CWI, called `apigen` [22]. Parsetrees produced by ASF+SDF Meta-Environment components are always represented by ATerms, so we can use the ATerm library to manipulate trees of any kind. `Apigen` generates an Abstract Programming Interface (API) to manipulate those trees without knowing that ATerms are the internal representation of them. This API is generated from SDF definitions. Both for AIL and AIL-Definition syntax, we have generated an API which is used by `bail` and `genailapi`.

But what is the binary format (ABF) of an AIL program? The assembler proceeds in a number of steps. First the AIL Definition argument is traversed and each instruction is mapped to an integer; that is, an opcode table is built. Then the AIL program is traversed and for each AIL statement appropriate action is taken. If the statement is a label, its position is saved[1]. Otherwise, if the statement contains an instruction, the signature is checked by looking up the instruction in the AIL definition. The program aborts if there is an error. If the statement is valid, the opcode corresponding to the instruction is appended to a binary buffer. If the instruction is followed by an argument, the argument is serialized to the binary buffer. This means that integers take up four bytes in the stream. ATerms are serialized with the `ATwriteToBinaryString` primitive provided by the ATerm library. If a label’s definition has already occurred the distance to the saved location is written to the stream. Labels thus correspond to relative byte offsets. If the definition of the label has not yet been encountered, a zero integer is appended to the stream. This kind of labels is resolved in a separate pass. Finally, the argument can consist of AIL code itself. In that case the byte compilation procedure is called recursively and the resulting bytecode program is appended to the stream.

[1]Note that label definitions in AIL are assumed to be just statements, but that they never take up space in bytecode.

6.2.4 Interpretation

As is mentioned above, the interpreter is partially generated by `genailapi`. By "partially" we mean that everything that is unrelated to the semantics of instruction is automatically derived from the AIL Definition. The interpreter that results is a so-called *threaded virtual machine*. This means that instruction branching is implemented using absolute addresses stored in an opcode table, instead of one while-loop containing a case statement which compares each byte of the instruction stream with some value. There are two main advantages of this approach. The first is speed: threaded virtual machines are generally much faster since no comparisons are needed. The second advantage is that multi-threading is much easier to implement. There is, however, one drawback: to implement a threaded interpreter in C, without falling back on some inline assembly, a GCC extension is used. Since GCC is available on many platforms, this is not a very high price to pay.

Operationally, the generated, threaded interpreter proceeds as follows. Let the array `opcodes` contain a reference to a label $\texttt{do}_i$ for each opcode i. So we have for all i: $opcodes[i] = \texttt{\&\&}\text{do}_i$. The unary operator `&&` is the aforementioned GCC extension and converts any label to a pointer to type `void`. Now the interpreter is defined as follows:

```
#define interpreter(x,program) {
  int_start: goto *opcodes[*program++];
    ∀ i ∈  range(opcodes) :
    do_i:  ⟨semantics of instruction with opcode i⟩
           goto *opcodes[*program++];
  int_end:
}
```

The interpreter starts by branching to the label associated with the first instruction opcode. Then for each kind of instruction, the following is generated. For instructions without arguments, just the implementing macro is invoked. If an argument is present, it is read from the byte stream (`*program`) according to its type and input to the implementing macro. If this argument is a label, the offset is converted to an absolute instruction location (pointer) using the current program position before it is input to the implementing macro. The size occupied by an argument in the bytestream is skipped. After the instruction dependent code has executed, the interpreter proceeds with the next instruction. The last label (`end`) of the should be used for termination. In the set of valid AIL instructions there should be one instruction which denotes termination. The implementing macro for this instruction can then branch to `end`. Thus, there is no support for implicit termination in AIL.

6.2.5 Compilation for Free

In this subsection we turn our attention to the (partially) generated compiler. The prototype macros that are to be filled in by the user, are used both by the interpreter and the compiler. We assume that these macros are properly implemented. Since the signatures of instructions are known from the AIL Definition, we can compile an AIL program to a C program that essentially consists of a sequence of invocations of the implementing macros.

Just as the interpreter, a compiled AIL program consists of a macro. This macro again accepts the unspecified “x” argument and a location to an AIL bytestream. This location, together with the interpreter appended to the sequence of compiled instructions, is used for executing reflective code. We will return to this subject later. The result of compiling an AIL program looks as follows:

```
#define compiled(x,pc) {
  declare_opcodes();
  cmp_start:
    ⟨instructions compiled to C⟩
  cmp_end:
  interpreter(x,pc);
}
```

The complete instruction sequence is surrounded by `cmp_start` and `cmp_end` labels which are used to distinguish static labels from dynamically produced labels.

The compilation of instructions is quite simple. For each kind of instruction we discuss the mapping to an equivalent in C. In the following we assume that *pos* designates the global position in the bytestream that is being compiled. This means, that *pos* is incremented with every byte that is consumed from the bytecode, even those in code that is argument to an instruction. We also assume that positions can be labels in C[2]. The compilation is specified as follows:

$$f \mapsto \text{`}pos\text{`} : \ \mathrm{f(x);} \tag{6.1}$$

$$f(n) \mapsto \text{`}pos\text{`} : \ \mathrm{f(x,}\text{`}n\text{`}\mathrm{);} \tag{6.2}$$

$$f(t) \mapsto \text{`}pos\text{`} : \ \mathrm{f(x,\ parse(}\text{`}unparse(t)\text{`}\mathrm{));} \tag{6.3}$$

$$f(l) \mapsto \text{`}pos\text{`} : \ \mathrm{f(x,\ \&\&(}\text{`}pos + l\text{`}\mathrm{));} \tag{6.4}$$

$$f(c) \mapsto \begin{array}{l} \mathrm{goto\ l_2;} \\ \mathrm{l_1} : \ \text{`}compile(c)\text{`} \\ \mathrm{l_2} : \ \text{`}pos\text{`} : \ \mathrm{f(x,\ \&\&l_1);} \end{array} \tag{6.5}$$

In the left column the various kinds of instructions are listed: instructions without arguments, and instructions with an argument of type integer, term, label and code respectively. The right column lists the C equivalent of the corresponding AIL instruction f. The expressions within ‘ (backquotes) are evaluated by the compiler. For example: ‘n‘ will result in an integer literal in C.

The first mapping is the most simple: an instruction f without arguments results in the invocation of the implementing macro “f” with the unspecified argument “x”. This argument “x” can be anything since macros are used to implement the semantics of an instruction. The argument “x” derives from the the toplevel macro in which the list of statements is embedded[3]. When the argument to f is an integer it is appended the list of arguments of the implementing macro. For ATerm arguments we first unparse the binary ATerm present in the stream to Textual ATerm Format (TAF, [7]). Converting an ATerm to TAF

[2]In the implementation these positions are translated into identifiers.
[3]Just as for the interpreter described above.

results in a textual representation *with sharing.* Thus, the argument is compiled to a function call which reads the ATerm from the textual representation (parse). Since byte offsets cannot be used in C, relative labels (offsets) are made absolute using the current position (*pos*) in the stream. This label always exist, since every instruction is labeled by its position in the stream. For code arguments, a goto statement is generated which jumps over the compiled code argument which is labeled by a new label l_1. The implementing macro gets this label (converted to a pointer) as an argument.

A Note on Reflection

In the interpreter code arguments are just AIL programs. That is, streams of bytes embedded in an ABF structure. It is up to the designer of language L what to do with such objects. However, when a binary AIL program is compiled we encounter a problem, since reflective arguments are themselves compiled to C. A code argument to an instruction will be a label pointer in compiled mode (see the last mapping above). It is up to the language designer what to do with this pointer. On the other hand, when some instruction, in one way or another, constructs an instruction sequence (bytecode) which should eventually be executed, it is not possible to compile this dynamically. This problem is solved by appending the interpreter as defined above, to the sequence of compiled instructions. Static code arguments are compiled and passed via label pointers,—dynamically created programs are interpreted via the included interpreter. Consider, for example, an AIL instruction `exec` which receives code in some way other than via code arguments (e.g. through an evaluation stack). Let's assume that both encodings are referenced via a pointer p[4]. This instruction can determine whether to branch to compiled code or to start the interpreter, by checking p as follows:

```
if (&&cmp_start < p ≤ &&cmp_end)
  goto *p;
else {
  pc = p;
  goto *opcodes[*pc++];
}
```

Labels can be disambiguated along the same line. Within the interpreter labels are made absolute using `pc` before passing it to the implementing macros. So by comparing this pointer with `&&cmp_start` and `&&cmp_end` we again know how to perform the jump.

6.3 Using AIL: an Example

The concepts and tools presented in the previous section are illustrated by defining a simple interpreter for arithmetic expressions in reverse polish notation (RPN).

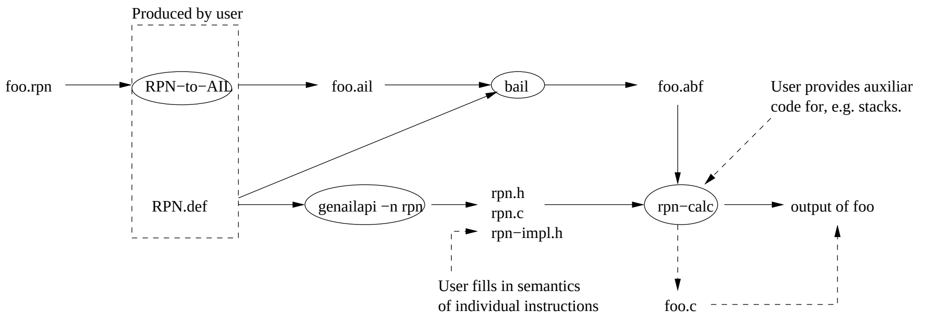


Figure 6.2: Interaction of AIL tools for RPN

6.3.1 Overview

To define an interpreter for reversed polish notation (RPN) using AIL a number of steps can be distinguished. These are depicted in Figure 6.2. First, a translation is needed from the RPN language to AIL. This step will not be elaborated any further in this section since it is trivial. Second, an AIL Definition should be present containing the allowed instructions together with their signatures. From this AIL Definition some C files are generated using `genailapi`. One of these files is to be modified by the user: `rpn-impl.h`. This file contains macro prototypes[5] for each instruction declared in the AIL Definition `RPN.def`. The semantics of each instruction can be specified in the bodies of these macros. The final step consists of writing a program (`rpn-calc`) that actually invokes the interpreter and contains all RPN specific machinery (such as an evaluation stack). A program `foo.rpn` in RPN notation can now be converted to an AIL program `foo.ail`. Using `bail` `foo.ail` is assembled to bytecode which is to be evaluated by `rpn-calc`. There is one aspect of Figure 6.2 that we have not yet mentioned. The program `rpn-calc` can also invoke the generated compiler. By doing this `foo.abf` is compiled to `foo.c`.

6.3.2 AIL Signature

For the bytecode compiler `bail` to know what kind of instructions are allowed, the signature of valid AIL instructions for RPN is defined as follows:

```
definition RPN {
        push <term>;
        add;
        sub;
        mul;
        div;
        output;
}
```

The only instruction that takes an argument is the `push` instruction which is used to evaluate constant values. The argument pattern `<term>` indicates that

[4]I.e. a `void*` pointer or an ABF pointer.

[5]Macro prototypes are just empty C preprocessor macros with the right number of parameters.

the argument can be any ATerm. This a very liberal constraint but more specific patterns could have been used just as easily.

6.3.3 Generated Code

The first part of the generated RPN API consists of a C header file (`rpn.h`) containing function declarations and macro definitions. The generation of these C entities depend solely on the AIL signature for RPN defined above.

```
#define RPN_push_opcode 0
#define RPN_add_opcode 1
#define RPN_sub_opcode 2
#define RPN_mul_opcode 3
#define RPN_div_opcode 4
#define RPN_output_opcode 5

#define RPN_num_of_instructions 6
#define RPN_lowest_opcode 0
#define RPN_highest_opcode 5

#define RPN_DECLARE_RPN_OPCODE_TABLE() \
static void *RPN_opcode_labels[] = {\
        [RPN_push_opcode] &&do_push,\
        [RPN_add_opcode] &&do_add,\
        [RPN_sub_opcode] &&do_sub,\
        [RPN_mul_opcode] &&do_mul,\
        [RPN_div_opcode] &&do_div,\
        [RPN_output_opcode] &&do_output\
}

#define RPN_goto_op(op) goto *(RPN_opcode_labels[(op)])
```

Since AIL programs are converted to a binary representation, an opcode table is needed. These opcodes are mapped directly to ordinary C labels. This mapping can be declared by invoking the macro `RPN_DECLARE_RPN_OPCODE_TABLE()`. The use of `&&` is a GCC extension and is used to convert any label to a void pointer. The declaration is encapsulated in a macro definition since labels can only be used in the scope where they are defined. To branch to a specific AIL instruction handler the second macro `RPN_goto_op` can be used. Note that the actual way of branching to an instruction handler is hidden from the user of this API.

```
#define RPN_interpreter(x,abf) {\
  RPN_start: RPN_goto_op(*abf);\
  do_push:\
    abf_skip_byte(abf);\
    {\
      ATerm d = abf_get_aterm(abf);\
      abf_skip_aterm(abf);\
      RPN_perform_push(x, d);\
    }\
    RPN_goto_op(*abf);\
  do_add:\
    abf_skip_byte(abf);\
    RPN_perform_add(x);\
    RPN_goto_op(*abf);\
  do_sub:\
    abf_skip_byte(abf);\
    RPN_perform_sub(x);\
    RPN_goto_op(*abf);\
  do_mul:\
    abf_skip_byte(abf);\
    RPN_perform_mul(x);\
    RPN_goto_op(*abf);\
  do_div:\
    abf_skip_byte(abf);\
    RPN_perform_div(x);\
    RPN_goto_op(*abf);\
  do_output:\
    abf_skip_byte(abf);\
    RPN_perform_output(x);\
    RPN_goto_op(*abf);\
  RPN_end: ;\
}
```

To further hide the branching mechanism an interpreter macro is generated which uses `RPN_goto_op` macro. This macro receives one unspecified argument x and a Abstract Binary Format structure which is the basis of AIL bytecode. Calling the interpreter defines the labels used in the opcode table. Since the signature of all AIL instructions is known from the RPN definition, any action pertaining to accessing the bytecode is generated. Thus, for the `push` instruction, we know that an ATerm follows the instruction byte, which is argument to the macro `RPN_perform_push`. This macro is declared empty in `rpn-impl.h`, and should be modified by the user. Interpretation continues after `RPN_perform_push` has finished.

The last declarations of `rpn.h` are function prototypes for the RPN compiler and disassembler:

```
char *RPN_getStringForOpcode(int op);
void RPN_disassemble(FILE *f, AIL_ByteCode bc);
void RPN_compile(FILE *f, AIL_ByteCode bc);
```

The definitions of these function are generated in `rpn.c`. The compile function for RPN simply constructs a C macro which contains the invocations of all instructions in the AIL bytecode argument. Per instruction, a label, a macro invocation and possibly an argument is printed. The function that compiles one instruction is defined as follows:

```
static void RPN_compile_instruction(FILE *f, ABF *abf, int
*line) {
  char op = abf_get_byte(*abf);
  fprintf(f, "l%d:  RPN_perform_%s", *line,
RPN_getStringForOpcode(op));
```

```
      *line += abf_byte_offset(*abf);
      abf_skip_byte(*abf);
      switch (op) {
        case RPN_push_opcode: {
          int len; char *s =
  ATwriteToSharedString(abf_get_aterm(*abf),&len);
          s = escape_quotes(s);
          fprintf(f,"(x,
  ATreadFromSharedString(\"%s\",%d));\\",s,len);
          *line += abf_aterm_offset(*abf);
          abf_skip_aterm(*abf);
          break;
        }
       default: fprintf(f,"(x);\\");
      }
      fprintf(f, "\n");
  }
```

As can be seen from this listing, for default instructions (no arguments) only the performing macro is invoked. For RPN there is only one instruction which accepts an argument: `push`. The argument is available in binary form in the bytestream `*abf` and is converted to a string containing the TAF respresentation. At runtime this string is parsed to reconstruct the ATerm. The result is input to the macro which implements the semantics of `push`.

6.4 AIL for Actions

6.4.1 Overview

One might argue that Action Notation is a low level functional language with side effects. This motivates the interpretation of actions as terms in a functional language. The compilation of actions to e.g. ML or Haskell thus seems obvious. The problem lies in the fact that actions are not just terms, but *very large* terms. The interpreter `evalan` and `acr` can handle these terms due to the maximal sharing in the representation by ATerms. In this section we elaborate a different approach by discarding the tree structure of actions altogether. Actions are serialized to a sequence of instructions which can be *executed* in the same way trees are *reduced.* Due to the SOS (small-step) nature of the semantics of AN2 reduction of terms is very costly in time. On the other hand the use of ATerms makes reduction very efficient in space. The AIL approach presented in this section is efficient in time, but in some circumstances (reflection) very expensive in space.

First the translation of some Full AN2 constructs to AIL is discussed. Then we describe the runtime architecture of `avm` in more detail than has been done in the overview chapter. Finally we discuss some implementation issues concerning special AIL instructions.

A_1 **then** A_2	$\mapsto$	$[[A_1]]$ `publish;` $[[A_2]]$ `unpublish;`
A_1 **and then** A_2	$\mapsto$	$[[A_1]]$ `push; copy;` $[[A_2]]$ `merge;`
A_1 **and** A_2	$\mapsto$	$[[A_1]]$ `push; copy;` $[[A_2]]$ `merge;`
indivisibly A	$\mapsto$	`lock;` $[[A]]$ `unlock;`
A_1 **exceptionally** A_2	$\mapsto$	`trye l1;` $[[A_1]]$ `catch l2:` `l1: epublish;` $[[A_2]]$ `unpublish;` `l2:`
A_1 **and exceptionally** A_2	$\mapsto$	`trye l1;` $[[A_1]]$ `catch l2:` `l1: epush;` `copy;` `trye l3;` $[[A_2]]$ `catch l4;` `l3: emerge;` `throw;` `l4: edrop;` `l2:`
A_1 **otherwise** A_2	$\mapsto$	`tryf l1;` $[[A_1]]$ `catch l2:` `l1: copy;` $[[A_2]]$ `l2:`
A_1 **hence** A_2	$\mapsto$	$[[A_1]]$ `enter;` $[[A_2]]$ `leave;`

Table 6.1: Translating Kernel AN2 to AIL

6.4.2 Translating AN2 to AIL

The compilation of actions to AIL instruction sequences is implemented in ASF+SDF in the very same way actions are compiled to C and Java. We will therefore not discuss the structure of this compiler. Moreover, since the reduction of Kernel actions has been discussed in the overview chapter (chapter 3), we here just list the translations of Kernel combinators in Table 6.1 as a reference. Instead, this subsection is devoted to describing the translation of some Full AN2 constructs. More specifically, we discuss the translation of the ***unfolding*** *A* and *unfold* actions, just as we did in chapter on `acc` and `ajc` (chapter 5). Again, we will distinguish the tail-recursive case from the non-tail-recursive case but we will see that some unexpected problems surface when tail-recursive actions are compiled to AIL in a straightforward way.

We start with the equations for ***unfolding*** *A* and *unfold* in the non-tail-recursive case.

```
[fs1]   is-unfolding(A) = true, get-unfolding(A) = A',
        is-tail-recursive(A') = false,
        env.pos = n,  n' = n * 10,
        lab1 = make-label(n), lab2 = make-label(n+1),
        env1 = env.unf := <lab1,false>,
        env2 = env1.pos := n'
        ===
        fa-to-stats(A, env) =
                frame lab2;
        lab1:   fa-to-stats(A', env2)
                return;
        lab2:
```

The first condition holds if A is the Kernel reduced form of ***unfolding*** A'. Since actions are reduced using an inner most reduction strategy we cannot use matching in the lefthand side of the equation[6]. The unfolded action is assigned to variable A'. If A' is not tail recursive, two new labels are generated using the position is the tree (`env.pos`). Next, the environment is updated at index `unf` with a tuple containing the first label and a boolean indicating A' is not tail recursive. After the position update of the environment with n', the AIL representation is produced containing the reduction of A'. The first instruction allocates a new frame with return address at `lab2`. Then the equivalent of A' is executed. Since `lab1` has been passed down the action tree via the environment, the reduction of *unfold* can use this information. *Unfold* first allocates yet again a new frame, then branches to `lab1` at the start of ***unfolding*** A. In the case that *unfold* in A' is not executed, control will eventually arrive at the `return` statement which is why the frame stack will be unwinded.

```
[fs2]   is-unfold(A) = true, env.unf = <lab, false>,
        env.pos = n, lab' = make-label(n)
        ===
        fa-to-stats(A, env) =
                frame lab';
                goto lab;
        lab':
```

To make the idea more clear let's consider the following action:

```
unfolding (copy and provide 0 then check _>_
   and then (copy and provide 1) then give _-_ then unfold
   exceptionally provide ())
```

This action loops until the given integer input becomes zero and then provides the empty tuple. Let's assume that this action is not tail-recursive. The translation to AIL using the above equations for ***unfolding*** A and *unfold* is depicted in Figure 6.3[7]. The crucial point of this action is the reachability of the `catch` instruction and the last `unpublish` instruction. They ensure that the data stack as well as the exceptional context stack are unwinded. The `frame` instruction at

[6]If we would like to match `fa-to-stats(unfolding A,env)`, some sort of *narrowing* would have been needed.

[7]Irrelevant instructions, such as type checks, are left out.

```
    frame l2;
l1: trye l3;
      copy;
      push;
      prov int(0);
      merge;
      publish;
        gt;
      unpublish;
      push;
      copy;
      push;
      prov int(1);
      merge;
      merge;
      publish;
        sub;
      unpublish;
      publish;
        frame l4;
        goto l1;
l4:   unpublish;
    catch l5;
l3:   prov [];
l5: return;
l2:
```

Figure 6.3: Example looping action in AIL

the beginning ensures that the last return that is executed returns to label `l2` which is the end of the loop. But before `return` jumps to `l2` the the destination is label `l4`. This happens n times (where n is the number of iterations) due to the allocation of frames preceding the `goto` instruction.

A control flow diagram is displayed in Figure 6.4. In this figure the input is assumed to be 3. The labels of edges denote the number of jumps. Label "2,1,0" corresponds to the consecutive values of the loop counter. Label "0,1,2" designates the undoing of stack operations that occurred during the decreasing of the loop value. Both the data stack and the context stack are popped. As can be seen from the control flow diagram, four frames are allocated and return is four times executed.

It is, however, obvious that the example action is tail recursive, so it should be possible to eliminate the instructions concerning the unwinding of stacks. Recall that by *tail recursive* we mean: the result of *unfold* is not used. If ***unfolding*** A is tail recursive, the `frame` instruction can be left out and action A is just prefixed with a label to be able to return to it with a `goto` instruction.

```
[fs1]   is-unfolding(A) = true, get-unfolding(A) = A',
        is-tail-recursive(A') = true, env.pos = n,
        n' = n * 10, lab1 = make-label(n),
        env1 = env.unf := <lab1,true>,
        env2 = env1.pos := n'
        ===
        fa-to-stats(A, env) = lab1: fa-to-stats(A', env2)
```

Now *unfold* becomes a jump to the label associated to ***unfolding*** A.

```
[fs2]   is-unfold(A) = true,
        env.unf = <lab, true>,
        env.pos = n
        ===
        fa-to-stats(A, env) = goto lab;
```

The example action, which *is* tail recursive, compiled using the last two equations, is listed in Figure 6.5. There are two problems with this compilation: first

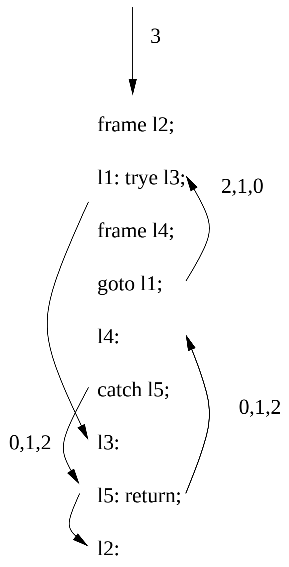


Figure 6.4: Control Flow of Example Action with Input 3

the value computed by the substraction is lost and second, the `catch` instruction is never reached (resulting in a corrupt context stack). The cause of the first problem is the `copy` instruction at the beginning which copies the original value (and not the decreased one) into the normal register. This can be solved by moving `copy` before label `l1`. It is however not clear how to do this for every case, since the copying may occur later and is needed when entering the loop. Another solution is the following: instead of using `publish` to make the decreased value available (just before the `goto` instruction) use an instruction that *replaces* the top of data stack with the value in the normal register. The input to the loop is replaced by the input to *unfold*. This way no `unpublish` instructions are needed (and thus no frame allocation), and the `copy` instruction is unaffected.

A possible solution to the second problem would be a `trye`-like instruction that does not save the context if an identical context[8] has already been saved. Then, the `catch` instruction would be redundant, and the only context that needs to be dropped will be dropped when the loop terminates,—that is, in case of an exception. Further research is needed to formulate the precise circumstances in which such a special `trye` instruction could be used[9].

[8]Identity is defined by equality of the handler labels.

[9]Suggestion: if *unfold* only occurs in the lefthand of the ***exceptionally*** combinator, and ***exceptionally*** occurs within an ***unfolding*** and is as close to *unfold* as possible (i.e. contains no other exceptional combinations).

```
l1:  trye l11;
       copy;
       push;
       prov int(0);
       merge;
       publish;
         gt;
       unpublish;
       push;
       copy;
       push;
       prov int(1);
       merge;
       merge;
       publish;
         sub;
       unpublish;
       goto l1;
     catch l12;
l11: prov [];
l12:
```

Figure 6.5: ‘Tail-recursive’ Version of the Example Action in AIL

Chapter 7

Conclusions

This chapter serves to present some conclusions. First we try to answer our research questions along three lines: use, engineering and implementation. At the same time we attempt to compare the various tools qua estimated implementation effort and usage scope. Then Section 7.2 puts the issue of the *use* of Action Semantics based tool generation in a wider context. Finally we present some future work.

7.1 Questions Answered

We have presented a number of tools for executing and compiling actions. During the design and implementation we have focussed mostly on software engineering aspects and the use in practical applications. It turned that the new Action Notation is a good starting point to generate interpreters and compilers for semantical descriptions of new programming languages, if performance is not the primary concern. There is however, one big proviso. The reduction of Full AN2 constructs to Kernel AN2 may be a great way of simplifying the semantics of AN2, but in some cases it puts a too great burden on the implementation of interpeters and compilers. Ironically, due to the reduction step a problem returns that AN2 was supposed to solve: computational concepts are hard to recover from the semantical representation of a program. For example, when looking at the reduction of ***unfolding*** *A*, it is hard to discern the concept of a loop from the (reflective) kernel representation. This state of affairs becomes even worse when one applies optimizations such as partial evaluation over the functional facet. Two additional technical points need to be emphasized in this context: the burden of bindings and the frequency of reflection. Since bindings are essentially finite maps and obey operations like disjoint union and overriding, the implementation can only be efficient when all bindings are eliminated at compile time. Binding elimination is, however, aggravated by the aforementioned problem of the reduction of Full AN2 to Kernel AN2. The problem with reflection has also to do with the reduction. Although we have implemented reflection in all of our tools, it should be said that it is not a cheap method for e.g. implementing loops or closures. Further research will be aimed at eliminating all of these unwanted cases of reflection. This means compiling more Full AN2 constructs like ***closure*** *A* directly.

Component	LoC/NoE/NoP	Notes
AN2	91 NoP	both Kernel and Full
`evalan`	125 NoE	includes reduction Full to Kernel
DN2	43 NoE	
ATerm DN	896 LoC	
`acr`	1181 LoC	
`acc` Runtime	1007 LoC	
`acc`	99 NoE	
AIL	20 NoP	
`avm` Runtime	640 LoC	
`avm` Arch	1003 LoC	partially generated by `genailapi`
`avm`	34 LoC	
`ailcc`	14 LoC	
`an2ail`	75 NoE	
ABF	105 LoC	
`bail`	1056 LoC	includes `genailapi`
AJC Runtime	275 LoC	
AJC DN	786 LoC	
`ajc`	104 NoE	

Table 7.1: Size Metrics for AN2 Components

In Table 7.1 we have listed size metrics for every component involved. For components that have been implemented in C or Java, the numbers denote lines of code (LoC). For syntax modules the metric is expressed in number of productions (NoP). Finally, ASF+SDF specifications are measured in number of equations (NoE)[1]. It should be clear that these statistics should not be taken for granted; a lot of code is generated. Some notes are therefore in order to assess the effort. For example, the effort for `avm` and `ailcc` seems impressively small in LoC, but one should realise it includes the effort for `bail`, ABF, the ATerm Data Notation and the `avm`-Runtime. On the other hand, we want to emphasize, that changes to the AIL instruction set are easy to make and that the AIL framework (including `avm` Architecture in this case) does not depend on Action Notation per se. It can be reused for other goals. In addition to this: the ATerm Data Notation is not only used by `avm` and `ailcc`, but also by `acc` and `acr`. A different kind of reuse was possible in the context of the ASF+SDF modules `ajc`, `acc` and AN2 to AIL. Although these modules do not so much as share code (except the AN2 syntax), their structure is practically the same: they all apply the scheme of 'layering' of the compilation function. Further research is needed to investigate the possibilities of further generalizing this architectural concept to allow the easy addition of other targets.

To answer the questions of the Introduction, Table 7.2 displays a qualitative comparison between the various tools that we have implemented. The advantages of `evalan` are manifold from the language designer's perspective. ASF+SDF is easily extensible and modular, which makes it a perfect match with AN2. One can define prototype languages in an iterative, incremental approach. Since `evalan` can be connected to the ASF+SDF Meta-environment

[1]Note that these specifications contain little or no syntax since we have separated semantics and syntax in all cases.

	Full Kernel	Data Notation	Performance	Ease of Use
`evalan`	√	ASF+SDF	−−	++
`acr`	√	ATerms	−	+
`avm`	√	ATerms	++	−
`acc`	×	ATerms	++	+
`ajc`	√	Classes	+	++

Table 7.2: Assessment of approaches to executing Actions.

it should be possible to have an environment specialized for Action Semantics Language Definition. We think that `evalan` is at its best when testing language definitions during the design phase of a language. Is is however too slow for real applications.

The term rewriter `acr` is two times faster but is again very close to the MSOS semantics of AN2. Since ATerms are used to represent abstract syntax of actions reflection is no problem. Using an embedded language like Guile [19], users can extend the interpreter in an adhoc way for any specific application. The Data Notation however, cannot be easily extended. We think `acr` is a better solution in a more stable phase of the language design cycle. We aim to integrate `acr` with the language independent debugger TIDE [37]. This should allow the debugging of programs on the level of the source language (abstracting away from the underlying action representation).

The AIL approach is interesting in that it translates a tree representation of an action to a sequential form. The speedup achieved with this approach is indeed substantial. However, there is a space tradeoff. Since reflection is still supported the construction of a new action using a combinator involves copying the streams of the argument actions; there is no sharing between sequences of instructions. Nevertheless the copying is needed to achieve generality: with it, dynamically constructed actions are still self-contained and can for instance be sent over a network connection without further ado. The drawbacks of the AIL approach are primarily concerned with use. First an action should be compiled to AIL which is subsequently byte compiled to a binary stream. This stream is then executed. The relation to the orignal action of source program is hard to establish and extension with new primitives or data operations is possible only with knowledge of how the AIL tools work.

With respect to extension the compiler to Java, `ajc`, provides a more user friendly interface. It should be possible to let a user annotate certain action primitives with a token which can be detected by `ajc`, and then compile that very action to a Java method call which is provided for by the user. This way the management of extensions is deferred to the moment of compiling the Java sources. The action to C compiler `acc` can deal with this in the same way. Furthermore the use of the Enactable interface allows the combination of actions with user defined semantics. Also, the platform independence of Java makes actions easily deployed. Platform indepedence is a sligthly more involved issue for `acc` and `avm`. Due to the use of the abstract JIT compiler Lightning, `acc` can only be used on Intel, Sun or PowerPC platforms,—that is, if reflection is used. In the case of `avm` there is only one dependency, which is the GNU C compiler (`gcc`) since instruction dispatch is implemented using computed gotos.

There is one important thing to note about the way actions are compiled to C and Java. Every combined action is compiled to a function resp. method declaration. This may not be the most efficient way of compiling actions, but is has a great pragmatic advantage: the origin of the original action is easy to establish. In fact it should be possible to even reconstruct the original action from the C/Java source code. Maintaining position information could eventually lead to informative error messages that contain the location in the source language code.

As can be seen, most tools heavily rely on the ATerm library. This might seem restrictive, but we deem that for the purpose of generating interpreters and compilers the Data Notation should *not* be a parameter of the Action Notation but should be provided in full strength. This allows communication between the tools and the interpreters become far easier to implement. The definition of AN2 states that bindings are data; so any implementation should have some structure that can address this problem. Using the ATerm library we have implemented bindings with bounded balanced trees to allow fast retrieval of bound values, while retaining transparency. Unless one is able to eliminate all bindings at compile time, bindings as data will be needed. Finally the interpretation of Actions themselves as data values is easy to integrate via ATerms, especially in the tree based approaches (`acr` and `evalan`). To allow further communication between different tools we are planning to implement a Java Data Notation using ATerms as well. The use of the Abstract Factory patterns alleviates the effort of doing this.

As a conclusion `ajc` and `acr` look the most promising from the user perspective and from the perspective of scope of use. The tools `ajc` and `acr` combine reasonable performance with easy extension and safety.

7.2 Discussion: What Use Is It Anyway?

There is an inherent ambiguity in the word *use*. The meaning of *use* swerves between the meaning of *application* and *benefit*. Both meanings should be accounted for. In this thesis we have certainly demonstrated that compilers and interpreters of Action Notation can be used in the first meaning of the word. The second meaning, however is not so clear. Unfortunately the *benefit* especially applies when considering the definition of Domain Specific Languages (DSLs). We do not aim to compete with handcrafted compilers. Therefore, the issue of DSLs naturally arises. However Action Semantics is designed without any particular language in mind. One could say AN2 is aimed at being the most general language there is. The problem we thus encounter is the *contradictio in terminis* in the sentence "using Action Semantics to define DSLs", since DSLs are never general, but highly specific (what's in a name?). So, as to the benefit of using Action Semantics to define DSLs, one could say the use of AN2 lies in the language architecture that one somehow obtains for free: scoping, control flow, exception handling, a store concept etc. However, the benefit depends on the way one is able to communicate to the outside world from within this language architecture. So the main problem is one of integration. That is the reason we have tried to concentrate on extensibility and communication.

7.3 Future Work

In this section some future work is discussed.

First we want to build a specialized Meta-Environment which connects a number of tools developed for this thesis. It is therefore needed that the tools can be connected to the Toolbus coordination architecture.

A different thread of future research is the elimination of the burdens of bindings and reflection. We want to investigate if it is possible to automatically arrive at more efficient implementations, although this is not a high priority. One of the options to achieve this is the definition of a type system for the Data Notation that is general enough to cope with most requirements.

Thirdly, it can be fruitful to generalize our compilation schemes and investigate whether it is possible to construct a generic compiler which is parameterized by the target language. It would then be easy to add new targets, like the .NET intermediate language (CIL) or C$^{\#}$. Different targets are of interest since the Java language is for some applications too restrictive. For example, Java has no parameterized types which prohibits the typing of cells and bindings. This is, by the way, a problem of all the tools that we have developed.

As a vision, future work will concentrate on trying to assess Action Notation as a kind of architectural pattern language which can be instantiated for specific domains. This amounts to make Action Notation domain independent. We aim to arrive at a certain kind of pluggable architecture which can be used to generate families of compilers or interpreters. The Feature Description Language (FDL) [11] can be used to describe the configuration of these families. On the basis of user requirements[2] a specialized interpreter or compiler can then be generated.

[2]Such language features might include: arbitrary precision arithmetic, side effects, non-deterministic choice, SQL access etc.

Appendix

Kernel AN2

The following table lists the complete kernel of AN2.

provide d	giving constant data
copy	copying given data
A_1 **then** A_2	functional composition
A_1 **and then** A_2	sequential composition
A_1 **and** A_2	interleaving
indivisibly A_2	anti-interleaving
raise	raising an exception
A_1 **exceptionally** A_2	exceptional composition
A_1 **and exceptionally** A_2	exceptional sequential composition
give o	computing dataoperations
check q	testing datapredicates
fail	abandoning an action
A_1 **otherwise** A_2	alternative composition
select $(A_1$ **or** ... **or** $A_n)$	nondeterministic choice
choose natural	arbitrary choice
give current bindings	current bindings as data
A_1 **hence** A_2	scoping of bindings
enact	performance of a given action
create	allocating and initializing a cell
destroy	deallocating a cell
update	updating a cell
inspect	inspecting a cell
activate	activating a new agent
deactivate	deactivating an agent
give current agent	current agent as data
send	sending a message to an agent
receive	receiving a message from an agent
give current time	current time as data

Full AN2

The Full AN2 level introduces the two new sorts *Yielder* and *Enquirer* which obey the follwing syntax:

```
Data | DataOp          -> Yielder
```

```
DataOp Yielder          -> Yielder
"(" {Yielder ","}2+ ")" -> Yielder
DataPred                -> Enquirer
DataPred Yielder        -> Enquirer
```

Yielders allow applicative composition of data operations. Enquirers allow data predicates to be applied to yielders.

the s, **a** s, **an** s	projection of input to sort s
it	equal to **the** datum
give Y	perform the action corresponding to yielder Y
A Y	function composition of action A and yielder Y
given Y	test input to be equal to Y
when E	test the holding of an enquirer
skip	neutral action (do nothing)
err	raise an exception with the empty tuple as data
tentatively A	execute A and fail when A raises an exception
infallibly A	execute A and raise an exception on failure of A
give (current bindings)	give current bindings
bound to Y	a yielder that retrieves the binding for Y
closure Y	a yielder ensuring static bindings for Y
bind	produce a binding
furthermore A	overriding non-local bindings
A_1 **moreover** A_2	overriding local bindings
A_1 **before** A_2	accumulating bindings
recursively A	recursively overriding bindings
unfolding A	allowing self-reference of A
unfold	perform action self-reference
stored in Y	a yielder that dereferences the cell yielded by Y
give (current agent)	give current agent
give (current time)	give current time
patiently A	repeating a failing action

Bibliography

[1] Stephen Adams. Implementing sets efficiently in a functional language. Technical report, University of Southampton Department of Electronics, 1992.

[2] J.C.M. Baeten and C. Verhoef. Concrete process algebra. Computing Science 3, Eindhoven University of Technology, January 1995.

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

[4] P. Bonzini. GNU Lightning. `http://www.gnu.org/software/lightning/`.

[5] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[6] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Annotated terms for efficient data exchange. *Xootic Magazine*, 9(2):20–30, November 2001. `http://www.win.tue.nl/xootic`.

[7] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

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

[9] Hermano Perrelli de Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, 1993.

[10] A. van Deursen and P. Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, 10:75–92, 1998.

[11] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[12] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notice*, 35(6):26–36, June 2000.

[13] A. van Deursen, P. Klint, and J. Visser. *Domain-specific Languages*, volume 28, pages 53–68. Marcel Dekker, Inc. New York, 2002.

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

[15] Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

[16] Erlang website. `http://www.erlang.org/`.

[17] J. Field, J. Heering, and T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys (CSUR)*, 30(3es):2, 1998.

[18] E. Gamma, Helm R., R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[19] GNU Guile website. `http://www.gnu.org/software/guile/guile.html`.

[20] P. H. Hartel. LETOS – a lightweight execution tool for operational semantics. *Software—practice and experience*, 29(15):1379–1416, Sep 1999. `http:// www.ecs.soton.ac.uk/ ~phh/ letos.html`.

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

[22] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. SEN 12, CWI, 2002.

[23] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

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

[25] Serge Lidin. *Inside Microsoft .NET IL Assembler.* Microsoft, 2002. Architectural Reference.

[26] Pierre-Etienne Moreau. A choice-point library for backtrack programming. In *Implementation Technology for Programming Languages based on Logic*, pages 16–31, 1998.

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

[28] Peter D. Mosses. Unified algebras and abstract syntax. Technical Report 94-5, BRICS, 1994.

[29] Peter D. Mosses. Theory and practice of action semantics. Technical Report 96-53, BRICS, 1996.

[30] Peter D. Mosses. CASL: a guided tour of its design. Technical Report 98-43, BRICS, 1998.

[31] Peter D. Mosses. Foundations of modular SOS. Technical Report 99-54, BRICS, 1999.

[32] Peter D. Mosses. AN-2: Revised action notation–syntax and semantics, 2000. `http://www.brics.dk/ pdm/papers/Mosses-AN-2-Semantics/`.

[33] Peter D. Mosses. CASL and action semantics. In Mosses and de Moura [36].

[34] Peter D. Mosses, editor. *AS2002*, Kopenhagen, Denmark, 2002. BRICS, Dept. of Computer Science, Univ. of Aarhus.

[35] Peter D. Mosses. What use is formal semantics? preliminary version, May 2002.

[36] Peter D. Mosses and Hermano Perrelli de Moura, editors. *AS2000*, Recife, Brazil, 2000. BRICS, Dept. of Computer Science, Univ. of Aarhus.

[37] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* PhD thesis, University of Amsterdam, 2000.

[38] Peter Ørbæk. OASIS: an optimizing action-based compiler generator. In *Proc. 5th Intl. Conf. on Compiler Construction (CC '94)*, volume 786 of *Lecture Notes in Compute Science*, pages 1–15. Springer-Verlag, 1994.

[39] GNU Sather website. `http://www.gnu.org/software/sather/`.

[40] T. van der Storm. AN2 tools. In Peter D. Mosses, editor, *Proc. 4th Intl. Workshop on Action Semantics (AS2002)*, pages 23–43, 2002.

[41] David Stoutamire and Stephen Omohundro. Sather – language specification. `http://www.gnu.org/software/sather/`.

[42] Arie van Deursen and Peter D. Mosses. ASD: The action semantic description tools. In *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582. Springer-Verlag, 1996.

[43] Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, Amsterdam, 1997.

[44] David A. Watt. JOOS action semantics. draft version 2.0, 2000. `http://www.dcs.gla.ac.uk/ daw/publications/JOOS2.ps`.