Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 167–188
www.elsevier.com/locate/entcs

An Action Compiler Targeting Standard ML

Jørgen Iversen

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

Abstract

We present an action compiler that can be used in connection with an action semantics based compiler generator. Our action compiler produces code with faster execution times than code produced by other action compilers, and for some non-trivial test examples it is only a factor two slower than the code produced by the Gnu C Compiler. Targeting Standard ML makes the description of the code generation simple and easy to implement. The action compiler has been tested on a description of the Core of Standard ML and a subset of C.

Keywords: Compiler generation, action semantics, code generation, Standard ML

1 Introduction

Automatically generating a compiler from a formal description of a language does not always lead to efficient compilers. A formalism that supports easy construction of readable, complete, and reusable descriptions of most programming languages and at the same time has tool support for automatically generating efficient compilers seems to be non-existing. One formalism that tries to satisfy these requirements to a language description formalism and allows automatic generation of efficient compilers is Action Semantics (AS) [12,16]. By efficient compilers we mean compilers that produce fast code, and not compilers that run fast or produce small code. An AS based compiler generator produces a front end which maps each program in the described language to

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

doi:10.1016/j.entcs.2005.02.057

an action. The front end is then connected to an *action compiler*, and the result is a compiler for the described language. Previous results [20,21] have shown that it is possible to generate compilers that produce code that is less than ten times slower than the code generated by handwritten compilers, and in some cases even as fast as only two times slower. Some restrictions have been put on the actions handled by the compiler to achieve this result, and often the implementation of the code generator in the action compiler is very complicated.

We present an action compiler that produces more efficient code than previous action compilers, and on some examples only a factor two slower than the code produced by the Gnu C compiler. The code generator translates actions to Standard ML (SML) [15] in a straight forward way. The SML code is then compiled to an executable using the MLton [2] compiler.

An action compiler annotates and transforms the action in several steps. Our action compiler performs type inference (Section 3) and code generation (Section 4), but no optimizations on the action as seen in previous results. Instead we generate code that can easily be optimized by MLton.

It is an advantage to be familiar with AS and SML, but not a prerequisite, when reading the paper. We will shortly introduce action semantics in the following section.

1.1 Action Semantics

Action Semantics (AS) is a hybrid of Denotational Semantics and Operational Semantics. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations, which model their behavior. The difference is that here denotations are *actions* instead of higher-order functions.

An Action Semantic Description (ASD) of a programming language must describe the syntax of the language, semantic functions mapping the language constructs to actions, and semantic entities used in the semantic functions. ASDs of non trivial languages, like Java [6] and SML [11], have already been constructed.

Actions are expressed in Action Notation (AN) [12,16], a notation resembling English but still strictly formal. AN consists of a *kernel* that is defined operationally; the rest of AN can be reduced to kernel notation. Actions are constructed from yielders, action constants, and action combinators, where yielders consist of data, data operations, and predicates. Yielders are not part

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

of the kernel.

The performance of an action might be seen as an evaluation of a function from data and bindings to data, with side effects like changing storage and sending messages. We shall often refer to the input data/bindings of an action as the *given data/bindings*. The action combinators correspond to different ways of composing functions to obtain different kinds of control and data flow in the evaluation. The evaluation can terminate in three different ways: *Normally* (the performance of the enclosing action continues normally), *abruptly* (the enclosing action is skipped until the exception is handled), or *failing* (corresponding to abandoning the current alternative of a choice and trying alternative actions). AN has actions to represent evaluation of expressions, declarations, abstractions, manipulation of storage, and communication between agents. The yielders can be used to inspect memory locations and compute data and bindings.

To limit this paper, we are not concerned with the actions used to represent communication between agents. Table 1 presents all kernel action combinators and constants, together with a short informal explanation. In the figure A ranges over actions.

Action	**Explanation**
copy	returns the given data
result D	returns data D
give O	applies data operator O to the given data
A_1 then A_2	output from A_1 is input to A_2
A_1 and-then A_2	sequencing, results are concatenated
A_1 and A_2	interleaving, results are concatenated
indivisibly A	A cannot be interleaved with other actions
check O	terminates abruptly if O returns $false$
choose-nat	returns a random non-negative integer
unfolding A	iterates A (in combination with unfold)
unfold	performs action A of smallest enclosing unfolding A
throw	terminates abruptly with the given data as result
A_1 catch A_2	A_2 receives output if A_1 terminates abruptly
A_1 and-catch A_2	abrupt sequencing, results are concatenated
fail	fails
A_1 else A_2	A_2 is the alternative if A_1 fails
copy-bindings	returns current bindings as data
A_1 scope A_2	the scope of bindings produced by A_1 is A_2
recursively A	allows recursive bindings in A
apply	applies the given action to the given data
close	computes the closure of the given action
create	allocates a fresh location
inspect	inspects the contents of the given location
update	updates the given location with the given data

Table 1
Kernel AN

Fig. 1 gives an example of an action. In line 1 a new memory location l_1, containing a random non-negative integer, is allocated. In line 3 the identifier

"x" is provided, and the action combinator in line 2 makes sure that line 3 is performed after line 1 and that the output from both evaluations is concatenated into the tuple (x, l_1). Line 4 passes the tuple to the action in line 5 which applies the data operator binding to it and returns the bindings map { x : l_1 }. The scope of these bindings is line 7 where they are just returned as data.

```
(1)       (((result x)
(2)      and-then
(3)        (choose-nat then create))
(4)    then
(5)      (give binding))
(6)  scope
(7)      copy-bindings
```

Fig. 1. Example of an action

1.2 Overview

In Section 2 we present the Action Environment which serves as a front end generator in our compiler generator. Type inference of actions is an essential part of generating efficient code from actions, and the subject of Section 3. The main contribution of this paper, namely the rules for translating actions into SML, is described in Section 4. Before evaluating the action compiler in Section 6 we take a look at previous work on compiling actions in Section 5. In Section 7 the limitations of our action compiler are discussed. Section 8 concludes.

2 The Action Environment

The Action Environment [4] is a tool for working with ASDs of programming languages. It supports the formalisms ASF+SDF [1,8] and ASDF [4,11]. The concrete syntax of a programming language can be described using arbitrary context-free grammars expressed in SDF. Abstract syntax in prefix constructor form and the action semantics of each construct can be described using ASDF. For mapping the concrete syntax to abstract syntax, ASF can be used. Using both ASF+SDF and ASDF a mapping from a language's concrete syntax to actions can be described. Fig. 2 shows how a program is transformed to an action using the Action Environment and a specification of a language. The action is then translated to SML using the action compiler and the ASDF part of the specification.

As explained in [4], the two formalisms have already been used to describe the core of the Standard ML language.

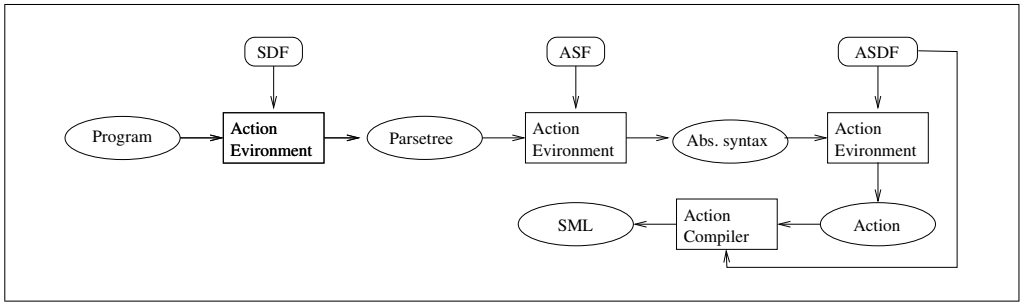

Fig. 2. The transformation of a program

In the environment it is possible to export both parse tables, for use in connection with a stand alone parser, and equations, describing the mapping from concrete syntax to actions, that can be used by an ASF evaluator. This makes it possible to map a program to an action independently of the Action Environment.

The Action Environment is the compiler front-end generator in our compiler generator (the front end consists of lexical analysis, parsing and transformation to an intermediate language, which in our case is AN). Connecting the front end with an action compiler, we have a compiler for the described language.

3 Type inference

Inferring a type for an action serves two main purposes. The first purpose is to type check the action. If a type can be inferred, we say that the action is type correct. If an action is type correct, it is guaranteed that during the evaluation of the action no sub-actions are given data or bindings of an unexpected type. As an example, the action "result true then give +" is not type correct because the sub-action "give +" expects two numbers but is given a boolean. By ensuring that the action is type correct no runtime type checks are needed, and this improves the efficiency of the code generated from the action.

Another purpose is to provide information about the runtime behavior of the action for use in code generation. The action and all its sub-actions are annotated with action types and, as we shall see later, some code generation rules use this type information. The type inference engine is described in [10].

The set of action types is described in Fig. 3. Action types — the types derived from the nonterminal *ActionType* — are function types where the domain is a pair of record types describing the type of data and bindings given to the action. We use a record type where the labels are numbers to describe product types, i.e., {1 : integer, 2 : boolean} corresponds to integer * boolean. The co-domain of the action type is a pair consisting of two record

$$
\begin{array}{lll}
\mathit{ActionType} & ::= & (\mathit{RecordType},\ \mathit{RecordType}) \rightarrow \\
 & & (\mathit{RecordType},\ \mathit{RecordType}) \\
\mathit{RecordType} & ::= & \{\ \mathit{Label} : \mathit{Type}\ , \ldots , \mathit{Label} : \mathit{Type}\} \mid \emptyset \\
\mathit{Type} & ::= & \mathsf{integer} \mid \mathsf{boolean} \mid \mathsf{token}(\mathit{Label}) \mid \mathsf{cell}(\mathit{Type}) \mid \\
 & & \mathit{ActionType} \mid \mathit{RecordType} \mid \ldots
\end{array}
$$

Fig. 3. Action types

types which describe the type of data produced by the action in case of normal or abrupt termination. If an action cannot terminate normally or abruptly, the special record type $\emptyset$ is used to indicate this. The action type

$$(\{\}, \{\ \mathsf{x} : \mathsf{cell}(\mathsf{integer})\}) \rightarrow (\{1 : \mathsf{integer},\ 2 : \mathsf{token}(\mathsf{a})\ \}, \emptyset)$$

describes the actions that expect no data, and a binding of "x" to an integer memory cell. The actions can terminate normally producing a pair consisting of an integer and the token "a" [3]. The record type $\emptyset$ tells us that the actions cannot terminate abruptly. Actions can have types that indicate that they might both terminate normally and abruptly, but of course not in the same evaluation; the types just say that they will terminate in one of the two ways.

The types derived from the nonterminal *Type* contain atomic types, like integer, boolean and token(*Label*). It also contains record types, because actions can produce bindings as output data, and action types, because actions can be treated as data. There are more types than the ones listed here, including types defined by the user in the ASDF specification. The type inference engine also uses information from the ASDF specification to determine the type of data and data operators.

The action "(result 5 then throw) catch fail" can be annotated with the following types (we use @ to separate a sub-action from its type):

$$
\begin{array}{l}
(((\mathsf{result}\ 5\ @\ (\{\}, \{\}) \rightarrow (\{1 : \mathsf{integer}\}, \emptyset)) \\
\mathsf{then\ throw}\ @\ (\{1 : \mathsf{integer}\}, \{\}) \rightarrow (\emptyset, \{1 : \mathsf{integer}\})) \\
\quad @\ (\{\}, \{\}) \rightarrow (\emptyset, \{1 : \mathsf{integer}\}) \\
\mathsf{catch\ fail}\ @\ (\{1 : \mathsf{integer}\}, \{\}) \rightarrow (\emptyset, \emptyset)) \\
\quad @\ (\{\}, \{\}) \rightarrow (\emptyset, \emptyset)
\end{array}
$$

How the type information is used will be explained in Section 4. Knowing that we translate actions to SML and that the SML compiler does type inference, one might wonder whether this type inference is necessary. If the SML code can be produced without knowledge of types, the SML compiler could try to infer a type and thereby check that the input action is type correct. The problem with this approach is that the generated SML code would be less efficient if the code generator could not take advantage of type annotations. To give an example, the translation of the and combinator (Rule 4 in

[3] The types describing tokens are very fine grained because knowledge about the specific token value is needed when inferring the type of bindings used by an action.

Section 4) uses knowledge about the size of the data tuples produced by its sub-actions. If this knowledge was not available data tuples would have to be represented by lists, which is less efficient.

4 Code generation

We have chosen to use SML as the target language for our action compiler. Previous work has used SPARC assembler code [21], C [7,5], Java [13,5], and a tailor made bytecode language[5] (see Section 5), but we found the translation from AN to SML more natural due to AN's resemblance with functional languages. The formal semantics of SML should make it relatively easy to prove that the produced code is semantically equivalent to the target action.

The translation is described using conditional rules, some of which are shown in Figs. 4–8, and the rest in the appendix. An action is inductively translated to SML by translating its sub-actions and then combining the produced code such that it captures the semantics of the action (Rule 3 illustrates this). Every action is translated into an anonymous function on the form "`fn (t, b) =>` E", where `t` is the data and `b` the bindings given to the action. The expression E computes the result of applying the function, which corresponds to the data produced when evaluating the action.

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

4.1 Flow of control and data

The action copy has the simplest translation (Rule 1) since it just returns the data given to it. Translating result D is only a little bit more complicated (Rule 2); here the data D produced by the action must be translated into an SML expression E. If the data D is an action it is translated using the rules, in other cases it is translated into SML representations of the data, e.g., integers and booleans are just translated to the same integers and booleans.

Normal composition of actions, as described by the then combinator, naturally translates to composition of the translations, E_1 and E_2, of the two sub-actions. The result of applying E_1 to the given data and bindings is given

$$\texttt{copy} \longrightarrow \texttt{fn (t, b) => t} \tag{1}$$

$$\frac{D \longrightarrow E}{\textsf{result}\ D \longrightarrow \texttt{fn (t, b) =>}\ E} \tag{2}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2\end{array}}{A_1\ \textsf{then}\ A_2 \longrightarrow \texttt{fn (t, b) =>}\ E_2\texttt{(}E_1\texttt{(t, b), b)}} \tag{3}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ n_1 = |normout(\mathcal{T}(A_1))|,\ n_2 = |normout(\mathcal{T}(A_2))|\end{array}}{\begin{array}{l}A_1\ \textsf{and-then}\ A_2 \longrightarrow \texttt{fn (t, b) =>} \\ \qquad \texttt{let val (}d_1, \ldots, d_{n_1}\texttt{) = }E_1\texttt{(t, b);} \\ \qquad\quad \texttt{val (}d_{n_1+1}, \ldots, d_{n_1+n_2}\texttt{) = }E_2\texttt{(t, b)} \\ \qquad \texttt{in (}d_1, \ldots, d_{n_1+n_2}\texttt{) end}\end{array}} \tag{4}$$

Fig. 4. Normal flow of control and data

to E_2 together with the same bindings used by E_1. Another solution would be to preface the generated code with a function "`fun asthen (A1, A2) (t,b) = A2(A1(t,b), b)`", and then translate the action to `asthen (` E_1, E_2`)`. A similar solution can be applied in some of the other code rules, namely the cases where the rules are not dependent on the type of the action. It will not change the execution time of the produced code, or make the translation remarkably easier, so to keep the uniformity of the code rules we have chosen the other translation.

The and-then combinator translates to a `let-in-end` expression (Rule 4). This expression can be used to describe declarations that are local to an expression. Both of the translations of the two sub-actions are evaluated on the given data and bindings, and the elements in the resulting tuples of data are bound to variables $d_1, \ldots, d_{n_1+n_2}$ which are then used in the resulting data tuple. When given an action type, the function *normout* returns the record type that describes the type of data produced by an action in case of normal termination. The $|\cdot|$ operator computes the size of the record, so a tuple pattern with the right number of d's can be generated. The cases where $normout(\mathcal{T}(A))$ is $\emptyset$ should be handled by other rules because it indicates that part of the action will never be evaluated. The rules can be found in the appendix.

The actions unfolding A and unfold (Rule 6 and 5 in Fig. 5) are used to describe iteration. The semantics of unfold is that it evaluates the action A in the nearest enclosing unfolding A. The translation of unfolding just binds the translation of A to the identifier `unf`, so the translation of unfold should just apply the function bound to `unf` to the given data and bindings. The function resulting from translating unfolding is the function bound to `unf`. Notice the

$$\texttt{unfold} \longrightarrow \texttt{fn (t, b) => unf (t, b)} \tag{5}$$

$$\frac{A \longrightarrow E}{\texttt{unfolding } A \longrightarrow \texttt{let val rec unf = } E \texttt{ in unf end}} \tag{6}$$

Fig. 5. Iterative control flow

use of the "`rec`" keyword which ensures that `unf` can be used from within E. It is not required that the `unfold` call is tail recursive (as it is the case in some previous work [7]), but the SML compiler is able to optimize the code in the cases where it is tail recursive.

$$\frac{\begin{array}{c} O \longrightarrow E \\ I = \mathit{excepid}(\{\}) \end{array}}{\begin{array}{l} \texttt{check } O \longrightarrow \texttt{fn (t, b) =>} \\ \qquad \texttt{let val ch = } E\texttt{(t, b)} \\ \qquad \texttt{in if ch then t else raise } I() \\ \qquad \texttt{end} \end{array}} \tag{7}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I = \mathit{excepid}(\mathit{abruptout}(\mathcal{T}(A_1))) \end{array}}{A_1 \texttt{ catch } A_2 \longrightarrow \texttt{fn (t, b) => (}E_1\texttt{ (t, b) handle } I \texttt{ et => } E_2\texttt{ (et, b))}} \tag{8}$$

Fig. 6. Abrupt control flow

Abrupt data flow in AN is translated to raising and handling SML exceptions (Fig. 6). If the result of applying a data operator O to the given data is the boolean value $false$, the action "`check` O" (Rule 7) terminates abruptly with no data. Because SML requires that exceptions are declared before being used, some preprocessing of the whole action is needed; for every unique occurrence of a record type representing the type of data produced by a sub-action that terminates abruptly a new exception is declared. The unique exception name tied to a record type is returned by the function *excepid* when it is applied to a record type. Since `check` does not produce any data when it terminates abruptly, the record type given to *excepid* is the empty record {}. In the generated code the SML keyword `raise` is used to raise an exception. If the result of applying the data operator (`ch`) is true, the given data (`t`) is the result.

For handling abrupt termination AN provides the action combinator `catch` (Rule 8). The code generated from it uses the SML keyword `handle` to capture the exception raised by the evaluation of the left hand side expression (E_1 `(t, b)`). The pattern "I `et`" on the right hand side of `handle` ensures that only the right exceptions are handled, and that the raised data is bound to the identifier `et`. The alternative when the first expression terminates abruptly

is to evaluate the second expression with the data raised and the original bindings. The type operator *abruptout* returns the type describing the data produced by an action in case of abrupt termination.

4.2 Bindings and storage

The actions concerned with scopes of bindings are shown in Fig. 7. The action copy-bindings (Rule 9) resembles the action copy and therefore the generated code is also similar. The only difference is that the result of evaluating it is the given bindings instead of the given data.

$$\text{copy-bindings} \longrightarrow \texttt{fn (t, b) => b} \tag{9}$$

$$\frac{A_1 \longrightarrow E_1 \quad A_2 \longrightarrow E_2}{A_1 \text{ scope } A_2 \longrightarrow \texttt{fn (t, b) =>} \ E_2\texttt{(t,} \ E_1\texttt{(t, b))}} \tag{10}$$

$$\frac{\begin{array}{c} A \longrightarrow E \\ \{i_1 : t_1, \ \ldots, \ i_n : t_{n_i}\} = \mathit{bindings}(\mathcal{T}(\text{recursively } A)) \\ \{1 : \{j_1 : t_1, \ \ldots, \ j_n : t_{n_j}\}\} = \mathit{normout}(\mathcal{T}(A)) \\ B = \{j_1 \texttt{ = r1, } \ldots\texttt{, } j_{n_j} \texttt{ = r}n_j\}/\{i_1 \texttt{ = ref b1, } \ldots\texttt{, } i_{n_i} \texttt{ = ref b}n_i\} \end{array}}{\begin{array}{l} \text{recursively } A \longrightarrow \\ \quad \texttt{fn (d, \{}i_1 \texttt{ = b1, } \ldots\texttt{, } i_{n_i} \texttt{ = b}n_i\texttt{) =>} \\ \quad \texttt{let} \\ \qquad \texttt{val rec rv1 = fn x => (rv1 x);} \\ \qquad \texttt{val r1 = ref rv1;} \\ \qquad \texttt{...} \\ \qquad \texttt{val rec rv}n_j \texttt{ = fn x => (rv}n_j \texttt{ x);} \\ \qquad \texttt{val r}n_j \texttt{ = ref rv}n_j\texttt{;} \\ \qquad \texttt{val \{}j_1 \texttt{ = ref f1, } \ldots\texttt{, } j_{n_j} \texttt{ = ref f}n_j\texttt{\} = E (t, } B\texttt{)} \\ \quad \texttt{in} \\ \qquad \texttt{r1 := f1; ...; r}n_j \texttt{ := f}n_j\texttt{; \{}j_1 \texttt{ = f1, } \ldots\texttt{, } j_{n_j} \texttt{ = f}n_j\texttt{\}} \\ \quad \texttt{end} \end{array}} \tag{11}$$

$$\text{create} \longrightarrow \texttt{fn (t, b) => ref t} \tag{12}$$

Fig. 7. Scopes of bindings

The same similarity can be seen when comparing the combinators scope (Rule 10) and then (Rule 3). The second sub-action A_2 in "A_1 scope A_2" uses the bindings produced by A_1 together with the original data, and this is reflected in the way the functions generated from the sub-actions are composed.

More interesting is the rule for recursively (Rule 11), and this is also the most complicated of the code generation rules. When evaluating the action "recursively A" the bindings produced by the sub-action A is also part of the bindings given to A. The bindings b_1 given to "recursively A" (in the rule it is $\mathit{bindings}(\mathcal{T}(\text{recursively}A))$) and the bindings b_2 produced by A are combined by letting b_2 override b_1, and the result is given to A. This allows recursive

declarations, like for instance recursive functions, in A.

To capture this relatively complex semantics, we use a trick where we bind every identifier in the domain of b_2 to a reference containing a "dummy" value, and every identifier in the domain of b_1, but not in the domain of b_2, is bound to a reference containing the value originally bound to the identifier. These bindings are then given to the code generated from A, which produces new bindings. Finally these bindings are used to update the references containing dummy values with the correct values. Using infinitely recursing functions as dummy value ensures that all functions can be stored in the reference because the reference will hold functions of type $\alpha \rightarrow \beta$, where α and β are type variables. In Rule 11 A is expected to generate bindings where actions are bound to identifiers, but if it binds other types of values, the dummy values used in the generated code should be changed to values with the same types as the bound values.

From the above, we see that looking up bound identifiers and creating new bindings in A must also take account of the use of references by dereferencing and creating references.

There are three actions to describe manipulation of storage, but only the one for allocating new memory cells is shown in Fig. 7. The generated code for create takes advantage of the builtin SML datatype for references. The constructor `ref` is used to construct a reference containing the data given to the function. For the two other actions, inspect and update, two other SML data operations on references, `!` and `:=`, are used to lookup the value stored in a reference and store a new value in an existing reference.

4.3 Actions as data

The biggest advantage in using SML as target language is in the translation of the actions related to actions as data. Here we exploit the fact that SML has higher order functions. When the data produced by "result D" is an action, it is useful that SML allows a function to return a function as result. For the action apply the generated code is a function that expects a function (d_1) together with some data $(d_2, \ldots, d_n)$ and then applies d_1 to the data $d_2, \ldots, d_n$ together with the empty record representing no bindings.

$$\frac{n = |normout(\mathcal{T}(\mathsf{apply}))|}{\mathsf{apply} \longrightarrow \texttt{fn ((}d_1\texttt{, ..., }d_n\texttt{), b) => }d_1\texttt{ ((}d_2\texttt{, ..., }d_n\texttt{), \{\})}} \quad (13)$$

$$\mathsf{close} \longrightarrow \texttt{fn (a, b) => fn (t, \{\}) => a (t, b)} \quad (14)$$

Fig. 8. Actions as data

The action close results in a function that both expects a function (the parameter a) and produces a function (`fn (t, {}) => a (t, b)`). The produced function expects no bindings and just applies the function a to the data and the bindings given to the whole function.

4.4 Data and data operators

AN contains a number of builtin data operators on integers and booleans that can trivially be translated to corresponding SML data operators. The builtin data operators on binding maps (operators for creating single bindings, looking up bindings, uniting binding maps, etc.) are translated into selection of elements from records and construction of records. To translate these data operators the type information about the given bindings is used. ASDF lets the user specify data and data constructors, and these are also translated into SML by the action compiler.

4.5 Example

To finish this section, we will give an example of the result of translating an action to SML. The action "(copy and (result 5 then create)) then apply" expects an action and then applies this action to a memory cell containing the integer 5. The translation is shown in Table 2 (we have added integer postfixes to some identifiers to improve readability, and inserted comments describing which sub-action a sub-expression originates from).

```
(fn (t1, b1) => (* then *)
  (fn ((d1, d2), b2) => d1 (d2, {})) (* apply *)
     ((fn (t3, b3) => (* and *)
         let val d1 = (fn (t4, b4) => t4) (t3, b3); (* copy *)
             val d2 = (fn (t5, b5) => (* then *)
                         (fn (t6, b6) => ref t6) (* create *)
                            ((fn (t7, b7) => 5) (* result 5 *)
                                 (t5, b5), b5))
                         (t3, b3)
     in (d1, d2) end)
     (t1, b1), b1))
```

Table 2
Example of generated code

Notice that the order of the sub-expressions representing sub-actions is reversed compared with the whole action, when the sub-actions are combined using then. The `let-in-end` expression is the translation of the sub-action with and as root, and here the results of evaluating its two sub-actions are bound to the identifiers `d1` and `d2` which are then combined into a pair; the result of the whole sub-action.

5 Related work

The Actress system [7] showed how to compile actions into C code. The compilation involved several action optimizations where the most important one was binding elimination. The system has been tested on a specification of a small imperative language called *Specimen*, and the running time of the generated C code for some programs has been compared to running times for implementations of the same programs in Pascal. This comparison shows that the generated C code is between a factor 5 to 28 slower than the compiled Pascal code. The rules describing the code generation are complicated because they use a set of variables to pass data between actions and must keep track of which variables are used and have been used by sub-actions.

Peter Ørbæk's OASIS [21] generated SPARC assembler code. This system applied several optimizations known from handwritten compilers, like constant propagation and tail recursion detection. In a comparison between programs compiled with a generated compiler for an imperative language *HypoPL* and equivalent programs written in C and compiled with GCC 2.4.3 (with full optimization), Ørbæk showed that the code from the generated compiler was between 1.5 to 4 times slower. Due to the low level of the target language, the code generation is complicated [4].

Continuing the work done by Brown, Moura and Watt on the Actress system, Kent D. Lee developed the Genesis system [13]. The systems have many similarities with respect to type inference and action transformations, but instead of generating C code, Genesis generates Java bytecode. One advantage of this is the portability of the generated code. As with the OASIS system, the low level target language makes code generation complicated, and special transformations of actions are needed. Lee does not present any evaluation of the generated code.

A somewhat different approach has been demonstrated by Bondorf and Palsberg in [2]. By writing an action interpreter in Scheme and applying the Similix partial evaluator, they were able to generate an action compiler that generates Scheme code. The advantage of this approach is that it is easier to write an action interpreter than an action compiler, and the hard work is done by Similix. Should AN change it is also easier to update an action interpreter than an action compiler. Their evaluation of the generated scheme code shows that it is almost 100 times slower than code generated by a hand written compiler.

Recently Tijs van der Storm [5] has shown a simpler approach to compiling

[4] Code generation is not well documented in [21], but the source code of OASIS can be downloaded at ftp://ftp.daimi.au.dk/pub/empl/poe/oasis-2.2.tar.gz

actions to C and Java. Comparing it with Actress and Genesis the compiler is simpler because it does not perform any type inference or optimizations. Instead of translating an action combinator to a sequence of statements, it translates it to a function that calls the functions representing the sub-actions. Because his compiler does not perform type inference, the code produced is not easily optimized by the C (or Java) compiler which is reflected in his test results. Van der Storm has only documented a test where he uses an action calculating fibonacci numbers (see Section 5 in [5]). The best result achieved when calculating the 20th fibonacci number is a running time of 0.8 seconds. To compare we have achieved a running time of 0.5 seconds for calculating the 33rd fibonacci number on slower hardware (Intel Pentium III 1 Ghz) than the hardware used by van der Storm (AMD XP 1800+). We were not able run his action compiler to better compare the two compilers.

There is a huge selection of compiler generators employing other formalisms than AS available. We shall mention two systems here that also seems to be popular outside academia, contrary to the AS based systems.

The Eli system [9] is based on attribute grammars. In addition to using attribute grammars the user can specify part of the compiler by "analogy" which means that the system has a large library of constructs used in common programming languages, so if the user wants scope rules similar to the ones used by Algol 60, he should just include the right module in the specification instead of writing it from scratch. The user can also specify part of the compiler by "solution" which means that he can write arbitrary fragments of C code that solves a problem. There are no examples in the literature of using Eli for implementing compilers for functional or object oriented languages, but a large set of real world imperative languages (Algol 60, C, Pascal) have been implemented completely or partly. In [14] the compiler for a Pascal-like language generated by Eli is compared with GCC, and the results show that the Eli generated compiler produces code that is approximately 35 % slower than the code produced by GCC.

In Gentle [18] the specification of a compiler is done in a logic programming language which is used in all parts of the specification. The specification language resembles Prolog but is more restricted and therefore the unification algorithm could be optimized. In [19] Vollmer reports that Gentle generates very efficient compilers with respect to compilation time, and that user experience shows that developing compilers in Gentle saves time compared to hand-coding compilers.

6 Evaluation of the action compiler

The action compiler has been implemented using ASF+SDF, a formalism that makes it easy to implement especially the code generation rules. The drawback is that it does not lead to fast executables, and therefore we are not going to compare the compilation times in this section, as it is done in related work.

We have tested the action compiler as part of our compiler generator, meaning that we have given the compiler generator two descriptions of programming languages and then compiled some test programs with the generated compilers.

The tests were run on a 3.0 GHz Intel® Pentium® 4 with 512 Kb cache and 1Gb RAM running Linux 2.4.20-31.9. The generated SML code from the action compiler was compiled using the MLton 20040227 compiler.

The first language we tested was the Core ML language as described in [11]. In Table 3 the test results are shown. The following test programs were used:

fibo uses a recursive function to calculate the 40th fibonacci number.

ackerman computes the Ackerman function on the integers 3 and 11.

fibo-while calculates the 40th fibonacci number using a while loop and references. The calculation is repeated 2 million times to reduce the significance of the program startup time.

length declares a list datatype, then constructs a list of length 100000 using a recursive function, and finally calculates the length using another recursive function.

church constructs the Church encoding of 10 million, and then transforms the Church encoding of the number back to an integer by applying the encoding to the increment function and 0.

The test programs exploit both the functional and the imperative aspects of the Core ML language. The second column shows the running time for the output from the action compiler. The third column shows the running time for the program compiled with the MLton compiler, and the last column shows how many times slower the output from the generated compiler is.

The result for the **fibo** program is quite satisfying, while the results for **ackerman**, **fibo-while**, and **length** are acceptable. The main reason for the slowness of the **fibo-while** and **length** programs is the way we represent data types (references and lists) in the produced SML code. Because of the way the semantics of function application in Core ML is described, the action representing the **ackerman** program is not tail-recursive, as the ML program is, and therefore the MLton compiler does a better job in optimizing the ML program than it can do on the code produced by our action compiler

Program	Generated compiler	MLton	Factor
fibo	4.80 s	2.77 s	1.7
ackerman	4.33 s	0.63 s	6.9
fibo-while	1.24 s	0.34 s	3.6
length	1.13 s	0.21 s	5.4
church	7.83 s	0.33 s	23.7

Table 3
CoreML running times

on this program. The problem with tail recursion is not noticeable in the recursive Fibonacci program (**fibo**), because the recursive function there is not tail recursive. In the AS description of Core ML the action representing a function is wrapped in a data type, and this is the main reason for the bad results when running the **church** test program which exploits higher-order functions.

The compiler generator has also been tested on a small subset of C. The subset includes simple expressions, assign-, if- and while- statements, statement blocks, variable declarations, and recursive functions. The values are integers and arrays of integers, but no pointers. The seven test programs are:

fibo computes the 40th fibonacci number using a recursive function.

ackerman computes the Ackerman function on the integers 3 and 11.

decrement contains four mutually recursive functions calling each other while decrementing an integer argument from 10 million to zero.

fibo-while calculates the 40th fibonacci number using a while loop. The calculation is repeated 50 million times to reduce the significance of the program startup time.

euclid is an implementation of Euclid's algorithm that finds the greatest common divisor of 37 and 1023. This is repeated 20 million times.

sieve is an implementation of the Sieve of Eratosthenes that finds the prime numbers between 1 and 2 million. This is repeated 10 times.

bubble is an implementation of the bubble-sort algorithm on an array of integers of length 32000.

Table 4 compares the running times for the output from the generated compiler and the programs compiled with GCC 3.3.2.

The results for all test programs except **ackerman** and **decrement** are quite satisfying. On these programs the generated compiler generates code that is only a factor two slower than code generated by GCC on the average.

Program	Generated compiler	GCC	Factor
fibo	2.81 s	1.50 s	1.9
ackerman	3.60 s	0.60 s	6.0
decrement	19.15 s	1.26 s	15.2
fibo-while	4.24 s	1.70 s	2.5
euclid	1.88 s	1.12 s	1.7
sieve	2.51 s	1.25 s	2.0
bubble	1.69 s	0.82 s	2.1

Table 4
miniC running times

We think that the generated compiler produces so slow code on the **decrement** program, compared to the GCC compiler, because of the way the recursively combinator is implemented, which seems to be particularly inefficient when the program contains mutually recursive functions. In the **ackerman** and **decrement** programs the problem with non-tail recursive actions appears again.

Comparing a compiler generated from a subset of C with a compiler for the whole C language is of course not fair. It is likely that the generated compiler will become less efficient when we extend the subset of C, especially if we allow more data than just integers and arrays of integers. Adding more features often means that the simple semantics of a construct is replaced by a more complex semantics, for instance, adding pointers and floats to the subset of C would mean that the semantic of + becomes more complicated because the operator should now be overloaded. On the other hand our compiler generates code that performs bounds checking on arrays, which the GCC compiler does not, which makes the generated compilers less efficient.

The test results in this section reveals that the efficiency of the generated compiler largely depends on the optimality of the action semantic description it is based on. It is also clear that the action compiler should be improved with respect to the implementation of the recursively combinator and the representation of data.

6.1 Comparison with OASIS

Comparing our result with the results achieved by others it is clear that our generated compilers are more efficient than all AS based systems, except OASIS, but probably less efficient than compilers generated by Eli.

We have not been able to use the Ørbæk's OASIS system ourself because it is based on outdated software and hardware we do not have access to.

Our comparison is therefore based on the numbers listed in Table 5 taken from Section 4.5 in [20]. The numbers show the running times for *HypoPL* (a subset of Pascal) programs compiled with a generated compiler (second column), the running times for a similar program compiled with GCC with full optimisation (third column), and how much slower the output from the generated compiler is (fourth column).

Program	**Generated compiler**	**GCC**	**Factor**
fibo-while	0.8 s	0.5 s	1.6
sieve	1.2 s	0.3 s	4.0
euclid	2.1 s	0.7 s	3.0
bubble	0.4 s	0.2 s	2.0

Table 5
OASIS running times

We have implemented equivalent programs in miniC and the running times are displayed in Table 4. It is difficult to compare our action compiler with OASIS for various reasons:

- We are not able to test OASIS.
- OASIS uses another AN based on an restricted version of the original AN and extended with extra features.
- Older hardware was used when measuring the running times for OASIS, so we can only compare the factor that its output is slower than GCC's output.
- The HypoPL language is a different from miniC. For instance, HypoPL allows neither functions with more than one argument nor mutually recursive functions.
- The results reported on OASIS are only with one decimal precision, so the factor might vary up to +/- 1 on some results.

All in all it is hardly fair to compare Ørbæk's results with ours. With this caveat we are going to try anyway. When comparing the numbers in the factor column in the last four rows of Tables 4 and the numbers in the factor row in Table 5, we notice that the output from our generated compiler is 2.1 times slower than gcc on the average, whereas OASIS's is 2.7 times slower. It would have been interesting to see how OASIS's generated compiler performs on the **ackerman** and **decrement** programs where we have significantly worser results.

7 Limitations

Both the use of action semantics as input and SML as output in our compiler generator puts some limitations on the languages that can be described. Action semantics cannot describe all language features, for instance, `call/cc` known from many functional languages cannot be described in a straight forward way. Our type inference algorithm puts further limitations on the set of actions accepted, for instance, actions originating from an ML program exploiting ML's let-polymorphism are not accepted. Finally the target language of the action compiler also limits the language features that can be described. The strict type system in SML means that it is difficult, if not impossible, to describe languages with subtypes.

Support for user defined data types is work in progress. Currently we support the data defined in the ASDF modules as part of a language description. It is only possible to describe languages where the user can define his own data types to some extent. The **length** program in Section 6 is an example of how the user can define a list data type in the Core ML language, but the description of data types in Core ML is not fully supported by the action compiler yet, and only works on some examples. The representation of data is the main reason for performance loss in the generated compilers.

8 Conclusion and future work

We have presented an action compiler that, compared to previous results, is a small improvement with respect to the efficiency of the generated code. Our main contribution is the simplicity of the code generation where we use SML as target language.

Future work includes investigating how to generate code that is easier for the SML compiler to optimize. Especially the way data is represented in the generated code needs improvement. Relaxing the restrictions put on actions would also improve the system. Improving the type inference algorithm such that it accepts a bigger set of actions, would allow more natural descriptions of languages, but here we are also limited by the target language (SML) being strongly typed.

It would be interesting to see our compiler tested on the full Standard ML language or another realistic language, instead of just a sublanguage. Previous work has also only been tested on small languages; so far it has not been investigated how well action compilers scale to handle realistic programming languages. We think that at the moment van der Storm's compiler is the compiler that has the best chance of handling actions originating from a realistic

language description because it can handle data that can be described using ATerms [3].

Using another target language is also worth investigating. There are compilers for Scheme and OCaml that on some examples produce faster code than MLton does on similar SML programs.

Acknowledgments Thanks to Peter D. Mosses for suggestions on how to improve the action compiler and this paper, and Fabricio Chalub and Janus Dam Nielsen for proofreading the paper (they are of course not responsible for any flaws in the paper). The anonymous referees also gave useful feedback that helped improve this paper. Thanks to Ane Bøndergaard for general support.

References

[1] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, chapter 1. Addison-Wesley, 1989.

[2] A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

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

[4] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. In G. Hedin and E. V. Wyk, editors, *Fourth Workshop on Language Descriptions, Tools and Applications, LDTA 2004, Barcelona, Spain, Proceedings*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[5] T. van der Storm. An-2 tools. In Mosses [17], pages 23–42.

[6] D. Brown and D. A. Watt. JAS: A Java Action Semantics. In *Proceedings of the Second International Workshop on Action Semantics, AS'99, Amsterdam, The Netherlands*, BRICS NS-99-3, pages 43–55. Dept. of Computer Science, Univ. of Aarhus, 1999.

[7] D. F. Brown, H. Moura, and D. A. Watt. Actress: An action semantics directed compiler generator. In *Fourth International Conference on Compiler Construction, CC'92, Paderborn*, LNCS Vol. 641, pages 95–109. Springer-Verlag, 1992.

[8] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing Vol. 5. World Scientific, 1996.

[9] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992.

[10] J. Iversen. Type inference for the new action notation. In Mosses [17], pages 78–98.

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

[12] S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *Proceedings of the Third International Workshop on Action Semantics, AS 2000, Recife, Brazil, May 15-16, 2000*, BRICS NS-00-6, pages 19–36. Dept. of Computer Science, Univ. of Aarhus, 2000.

[13] K. D. Lee. *Action Semantics-based Compiler Generation*. PhD thesis, Department of Computer Science, University of Iowa, 1999. http://www.cs.luther.edu/~leekent/papers/thesis.ps.

[14] A. Macedo and H. Moura. Investigating compiler generation systems. In *Proceedings of the IV Brazilian Symposium on Programming Languages, SBLP 2000*, pages 259–266. Brazilian Computing Society, 2000.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

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

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

[18] F. W. Schröer. *The Gentle Compiler Construction System*. R. Oldenbourg Verlag, Munich and Vienna, 1997. http://gentle.compilertools.net/BOOK.ps.gz.

[19] J. Vollmer. Experiences with Gentle: Efficient compiler construction based on logic programming. In J. Maluszynski and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming, PLILP 1991*, LNCS Vol. 528, pages 425–426. Springer-Verlag, 1991.

[20] P. Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Dept. of Computer Science, Univ. of Aarhus, 1993. ftp://ftp.daimi.au.dk/pub/empl/poe/oasis.ps.gz.

[21] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *Fifth International Conference on Compiler Construction, CC'94, Edinburgh, Proceedings*, LNCS Vol. 786, pages 1–15. Springer-Verlag, 1994.

Appendix: Remaining Code Rules

$$\frac{O \longrightarrow E}{\mathsf{give}\ O \longrightarrow E} \tag{15}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ n_1 = |normout(\mathcal{T}(A_1))|,\ n_2 = |normout(\mathcal{T}(A_2))| \end{array}}{\begin{array}{l} A_1\ \mathsf{and}\ A_2 \longrightarrow \texttt{fn (t, b) =>} \\ \quad \texttt{let val } (d_1, \ldots, d_{n_1}) \texttt{ = } E_1\texttt{(t, b);} \\ \quad\quad \texttt{val } (d_{n_1+1}, \ldots, d_{n_1+n_2}) \texttt{ = } E_2\texttt{(t, b)} \\ \quad \texttt{in } (d_1, \ldots, d_{n_1+n_2}) \texttt{ end} \end{array}} \tag{16}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ \emptyset = normout(\mathcal{T}(A_1)) \end{array}}{A_1\ \mathsf{and}\ A_2 \longrightarrow E_1} \tag{17}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ \emptyset \neq normout(\mathcal{T}(A_1)),\ \emptyset = normout(\mathcal{T}(A_2)) \end{array}}{\begin{array}{l} A_1\ \mathsf{and}\ A_2 \longrightarrow \texttt{fn (t, b) =>} \\ \quad \texttt{let val \_ = } E_1\texttt{(t, b);} \\ \quad\quad \texttt{val \_ = } E_2\texttt{(t, b)} \\ \quad \texttt{in () end} \end{array}} \tag{18}$$

$$\frac{\begin{array}{c} A_1 \longrightarrow E_1 \\ \emptyset = normout(\mathcal{T}(A_1)) \end{array}}{A_1\ \mathsf{and\text{-}then}\ A_2 \longrightarrow E_1} \tag{19}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ \emptyset \neq normout(\mathcal{T}(A_1)),\ \emptyset = normout(\mathcal{T}(A_2))\end{array}}{\begin{array}{l}A_1 \text{ and-then } A_2 \longrightarrow \texttt{fn (t, b) =>} \\ \qquad \texttt{let val \_ =} E_1\texttt{(t, b);} \\ \qquad\quad \texttt{val \_ =} E_2\texttt{(t, b)} \\ \qquad \texttt{in () end}\end{array}} \tag{20}$$

$$\frac{A \longrightarrow E}{\text{indivisbly } A \longrightarrow E} \tag{21}$$

$$\frac{I = excepid(abruptout(\mathcal{T}(\text{throw})))}{\text{throw} \longrightarrow \texttt{fn (t, b) => raise } I \texttt{ t}} \tag{22}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ \emptyset = abruptout(\mathcal{T}(A_1))\end{array}}{A_1 \text{ catch } A_2 \longrightarrow E_1} \tag{23}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I_1 = excepid(abruptout(\mathcal{T}(A_1))), I_2 = excepid(abruptout(\mathcal{T}(A_2))) \\ n_1 = |abruptout(\mathcal{T}(A_1))|,\ n_2 = |abruptout(\mathcal{T}(A_2))|\end{array}}{\begin{array}{l}A_1 \text{ and-catch } A_2 \longrightarrow \texttt{fn (t, b) => (}E_1\texttt{ (t, b)} \\ \qquad \texttt{handle } I_1\texttt{ (}d_1, \ \ldots, \ d_{n_1}\texttt{) => (}E_2\texttt{ (t, b)} \\ \qquad\quad \texttt{handle } I_2\texttt{ (}d_{n_1+1}, \ \ldots, \ d_{n_1+n_2}\texttt{)} \\ \qquad\quad \texttt{=> (}d_1, \ \ldots, \ d_{n_1+n_2}\texttt{)))}\end{array}} \tag{24}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ \emptyset = abruptout(\mathcal{T}(A_1))\end{array}}{A_1 \text{ and-catch } A_2 \longrightarrow E_1} \tag{25}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2 \\ I_1 = excepid(abruptout(\mathcal{T}(A_1))), \emptyset = abruptout(\mathcal{T}(A_2))\end{array}}{\begin{array}{l}A_1 \text{ and-catch } A_2 \longrightarrow \texttt{fn (t, b) => (}E_1\texttt{ (t, b)} \\ \qquad \texttt{handle } I_1\texttt{ \_ => (}E_2\texttt{ (t, b)))}\end{array}} \tag{26}$$

$$\text{fail} \longrightarrow \texttt{fn (t, b) => raise FAIL} \tag{27}$$

$$\frac{\begin{array}{c}A_1 \longrightarrow E_1 \\ A_2 \longrightarrow E_2\end{array}}{A_1 \text{ else } A_2 \longrightarrow \texttt{fn (t, b) => (}E_1\texttt{ (t, b) handle FAIL => }E_2\texttt{ (t, b))}} \tag{28}$$

$$\text{choose-nat} \longrightarrow \texttt{fn (t, b) => random ()} \tag{29}$$

$$\text{inspect} \longrightarrow \texttt{fn (t, b) => !t} \tag{30}$$

$$\text{update} \longrightarrow \texttt{fn ((c, d), b) => c := d} \tag{31}$$