The Aladin Abstract Machine

Gary Meehan
Department of Computer Science
University of Warwick
Coventry
UK, CV4 7AL
E-mail: Gary.Meehan@dcs.warwick.ac.uk

December 17, 1998

Abstract

The Aladin Abstract Machine (AAM) provides a completely abstract definition of a functional language. There are no primitives built into Aladin, instead primitives are intended to be programmed in *any* language, functional or imperative and imported into the AAM. In this report we develop an efficient operational semantics for the AAM using the original denotational semantics as our starting point. We then use this semantics to develop an implementation of Aladin, using the Java language, with the ability to write programs in an Aladin scripting language that we develop, and primitives written in C/C++, Java and Ginger.

1 Introduction

The Aladin Abstract Machine (AAM) [1] provides a completely abstract definition of a functional language. There are no primitives built into Aladin, instead primitives are intended to be programmed in *any* language, functional or imperative and imported into the AAM. These primitives could be simple functions like addition; more complex higher-order ones like *map* or *fold*; or even complete programs such as *grep* or *wc*.

As well as the lack of primitives, Aladin has two more major features not found in the majority of functional languages: the ability to specify the strictness of a function's arguments *and* results; and the use of streams for ordered I/O and real-time operations (which we shall not consider further in this report).

Unlike most other abstract machines for functional languages, such as the G-Machine [2, 3], the SECD machine [4], and the Three Instruction Machine (TIM) [5], the AAM is concerned only with the evaluation of programs and not their construction. Since Aladin is designed to import its primitives from *any* language, it would be inappropriate to tie the abstract machine down to one particular way of constructing programs.

It is this simplicty and the requirment for the user to specify the strictness of functions that gives Aladin its advantages. By keeping things as purely functional as possible, without any added complications, and by knowing what needs to be evaluated and what doesn't, we hope to be able to use Aladin to investigate several areas of functional languages such as parallel and partial evaluation, and the effects of strictness on the time and space requirements of functional programs.

In this report we shall develop an efficient operational semantics for the AAM, starting from the original denotational semantics and going *via* a denotational semantics which includes explicit sharing and updating. We then use this semantics to develop an implementation of Aladin, using the Java language, with the ability to write programs in an Aladin scripting language that we develop, and primitives written in C [6], C++ [7], Java [8] and Ginger [9].

2 The Denotational Semantics of the AAM

An Aladin program is designed to be either a data object, a function of an application of one program to another. Denoting our set of data objects as D and our set of functions as F then our set of programs, P is:

$$P ::= D \mid F \mid P\ P$$

Multiple-arguments to functions are curried and applications are left-associative. Hence for $p_1, p_2, p_3 \in P$ we have:

$$p_1\ p_2\ p_3 \equiv (p_1\ p_2)\ p_3$$

Functions may be written in any language of the user's choice. A function f of arity m is denoted as f^m. The meta-function @ is used to primitively apply f to its arguments by executing the code associated with f. The expression $f^m@(p_1, \ldots, p_m)$ denotes the result of the primitive application.

The user is required to specify the strictness of its argument and result which gives the user complete control over the evaluation order of programs. We adopt a slightly different definition to strictness that that used in other functional languages. We say that a function is strict in an argument if it is required that the argument is evaluated before the function is applied and lazy otherwise. A function returns a strict result if it is not required to evaluate the result, a lazy result otherwise. For a function of arity m we use the notation:

$$f :: \sigma_1 \times \ldots \times \sigma_m \to \rho, \quad \sigma_i, \rho \in \{s, l\}$$

to denote a function which is strict in its ith argument if $\sigma_i = s$ and lazy if $\sigma_i = l$, and strict in its result if $\rho = s$ and lazy if $\rho = l$.

2.1 The Original Evaluation Rules

The original evaluation rules used a meta-function, **Eval**, to return the result of evaluating a program. If we have a data object applied to any number of programs, then **Eval** simply returns the original program:

$$\textbf{Eval}[\![\ (d \in D)\ p_1\ \ldots\ p_n\ ]\!] = d\ p_1\ \ldots\ p_n \tag{1}$$

If we have a function applied to too few arguments, then we return the original expression, but we evaluate any strict arguments:

$$\begin{array}{l}\textbf{Eval}[\![\ (f :: \sigma_1 \times \ldots \times \sigma_m \to \rho)\ p_1\ \ldots\ p_{n<m}\ ]\!] = f\ q_1\ \ldots\ q_n \\ \quad \textbf{where} \\ \quad \begin{array}{rcll} q_i & = & \textbf{Eval}[\![\ p_i\ ]\!], & \textbf{if } \sigma_i = s \\ & = & p_i, & \textbf{otherwise} \end{array}\end{array} \tag{2}$$

If we have a function applied to too many arguments, then we evaluate the inner application, that is the function applied to the exact number of arguments it needs, then apply the result of this to the remaining arguments.

$$\begin{array}{l}\mathbf{Eval}[\![\ f^m\ p_1\ \ldots\ p_{n>m}\ ]\!] = \mathbf{Eval}[\![\ r\ p_{m+1}\ \ldots\ p_n\ ]\!] \\ \quad \mathbf{where}\ r = \mathbf{Eval}[\![\ f^m\ p_1\ \ldots\ p_m\ ]\!]\end{array} \tag{3}$$

Finally we have the case when we have a function applied to exactly the right number of arguments. We have to do three things: evaluate any strict arguments; apply the function using @; and evaluate the result if necessary.

$$\begin{array}{l}\mathbf{Eval}[\![\ (f :: \sigma_1 \times \ldots \times \sigma_m \to \rho)\ p_1\ \ldots\ p_m\ ]\!] = r \quad \mathbf{where} \\ \begin{array}{rcll} r & = & e, & \mathbf{if}\ \rho = s \\ & = & \mathbf{Eval}[\![\ e\ ]\!], & \mathbf{otherwise} \\ e & = & f@(q_1, \ldots, q_m) & \\ q_i & = & \mathbf{Eval}[\![\ p_i\ ]\!], & \mathbf{if}\ \sigma_i = s \\ & = & p_i, & \mathbf{otherwise}\end{array}\end{array} \tag{4}$$

3 The Denotational Semantics of the AAM with Explicit Updates

Implementations of functional languages use sharing and updating to avoid doing repeated work. Our first step towards an efficient operational semantics for the AAM is thus a denotational semantics of the AAM which makes explicit sharing and updating.

3.1 Variables and the Heap

We associate each program with a variable $v \in V$, changing our syntax of Aladin programs to accommodate this:

$$P ::= D \mid F \mid V \mid V\ V \tag{5}$$

Applications now involve the application of one variable to another and a program can also be a variable, that is an indirection to the actual value. The explicit naming of all parts of a program, in particular functions, means we can implement recursion directly.

These associations are defined in a mapping of variables to programs which we shall call the *heap*. A heap provides the context for the evaluation of a program, and thus a program evaluated with respect to one heap can yield a different result to the same program evaluated w.r.t. a different heap. The expression

$$\Gamma[x_0 \mapsto p_0, x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]$$

denotes that in the heap Γ variable x_i maps to program p_i where $i \in 0, \ldots, n$. We may occasionally use the heap as a lookup-function, that is:

$$\begin{array}{rcll}\Gamma\ x & = & p, & \mathbf{if}\ \Gamma[x \mapsto p] \\ & = & \perp, & \mathbf{otherwise}\end{array}$$

Since a program could be a variable we could have a chain of indirections:

$$\Gamma[x_0 \mapsto x_1, x_1 \mapsto x_2, \ldots, x_n \mapsto p \notin V]$$

In such cases we allow ourselves to 'short-circuit' the chain and write $\Gamma[x_0 \mapsto p]$. Note that the chain could in fact be a cycle:

$$\Gamma[x_0 \mapsto x_1, x_1 \mapsto x_2, \ldots, x_n \mapsto x_0]$$

Which is akin to writing something like:

```
foo = x
    where
        x = y
        y = z
        z = x
```

in Haskell. The result of evaluating any program which tries to access a variable in such a cycle will be $\perp$.

We will allow ourselves to abuse notation slightly and let $\Gamma \cup \Gamma'$ denote the heap Γ updated with the mappings in heap Γ' with any clashes being resolved in the favour of those defined in Γ':

$$\begin{array}{rcll} (\Gamma \cup \Gamma')\ x & = & p, & \textbf{if}\ \Gamma'[x \mapsto p] \\ & = & q, & \textbf{if}\ \Gamma[x \mapsto q] \\ & = & \perp, & \textbf{otherwise} \end{array}$$

In particular, $\Gamma \cup \{x \mapsto p\}$ denotes a heap where x is associated with p and all other variables are associated to the programs that they were in Γ. The heap is a global object and all changes to it are universal and hence an implementation of a heap can do updates destructively.

Since all programs are evaluated w.r.t. a heap, it follows that our primitive application meta-function must also execute with respect to a heap. The informal type of @ changes from:

$$P \times \ldots \times P \to P$$

to:

$$\mathit{Heap} \times V \times \ldots \times V \to (P \times \mathit{Heap}) \tag{6}$$

Note that while @ takes a heap and variables as arguments it returns a *program* and a heap (we need to return a heap as the function may want to create new objects in the heap).

3.2 The Evaluation Rules

Our evaluation philosophy changes from that used in the original semantics. Whereas before we returned a new expression which represented the evaluation, we now evaluate a program by updating the heap which provides the context for the program. The meta-function **E** provides the top-level interface to this procedure:

$$\mathbf{E}\ p\ \Gamma[x \mapsto p] = (\mathbf{U}\ \Gamma\ x)\ x \tag{7}$$

So we find the variable associated to the program we want to evaluate[1], update the heap with respect to this variable, and finally look up the value of the variable in the updated heap.

[1] Since a heap is not injective, there could be any number of variables associated with the same program but it doesn't matter which one we pick.

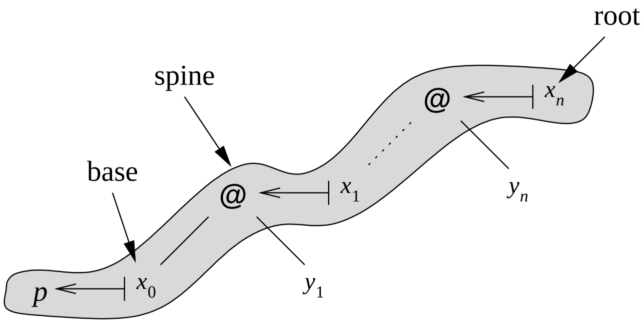

Figure 1: An generic unwound application

The updating of the heap is done by the meta-function **U** which updates a given heap by evaluating the program referred to by a given variable, which we shall refer to as the *root* of the program. If the root refers to an application then we need to traverse or unwind the *spine* (the shaded part of the figure) of the application until we reach a non-application. In the heap

$$\Gamma[x_0 \mapsto p, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]$$

if x_n forms the root of the program then the $x_i, i \in 0, \ldots, n$ form the spine and x_0 forms the base (see Figure 1).

The four rules for **U** (8–11) directly reflect those of **Eval** (1–4). Suppose we have a data object applied to a number of arguments. Then since no evaluation needs to be done, no updating of the heap has to be done either:

$$\mathbf{U}\ \Gamma[x_0 \mapsto d \in D, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_n = \Gamma \tag{8}$$

If we have a function applied to too few arguments then we just need to evaluate the strict ones. The **A** meta-function (see rule 13) returns the heap resulting from evaluating the strict arguments of a function.

$$\mathbf{U}\ \Gamma[x_0 \mapsto f^m, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_{n<m} = \mathbf{A}\ \Gamma\ x_n \tag{9}$$

If we have a function f^m applied to too many arguments, $x_1, \ldots, x_m$ say where $n > m$ (see Figure 2), then we first obtain the heap resulting from evaluating the inner application (the shaded part of the figure) the root of which is x_m. In this new heap, x_m will refer to the evaluated version of $f^m\ x_1\ \ldots\ f^m$, r say (see Figure 3). This result becomes the new base of our program (note that we may need to do some more unwinding if r is an application) and we continue updating.

$$\begin{array}{l}\mathbf{U}\ \Gamma[x_0 \mapsto f^m, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_{n>m} = \mathbf{U}\ \Gamma'\ x_n \\ \quad \textbf{where}\ \Gamma' = \mathbf{U}\ \Gamma\ x_m\end{array} \tag{10}$$

If we have a function f applied to the exact number of arguments, we first evaluate any strict ones using **A** and primitively apply the function using @. The root of the application then needs to be update with the result of @ and finally we may need to continue evaluation

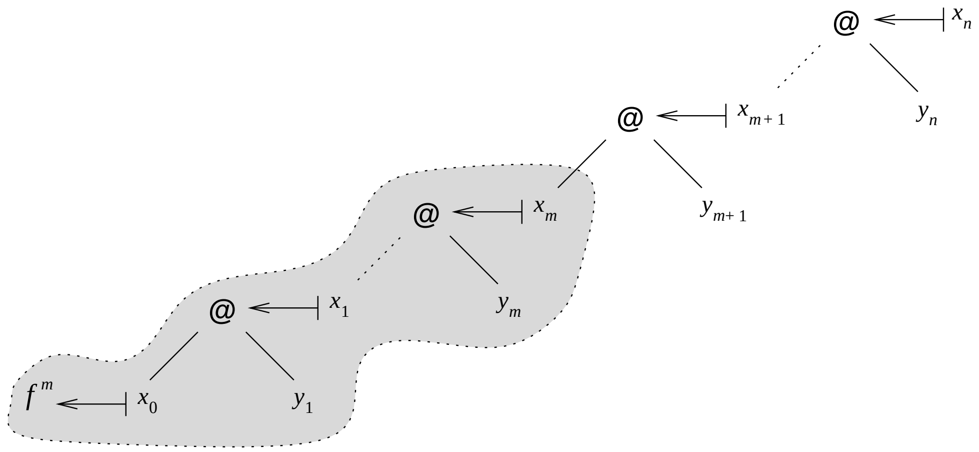

Figure 2: Application of a function to too many arguments

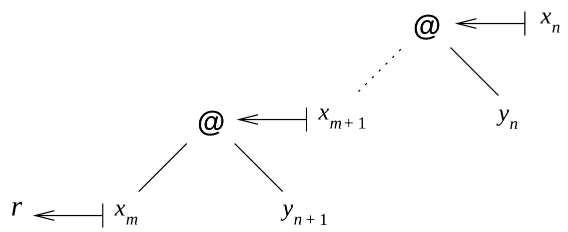

Figure 3: Application of a function to too many arguments after evaluation of inner application

if f returns a lazy result.

$$\begin{array}{l}\mathbf{U}\,\Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho, x_1 \mapsto x_0\ y_1, \ldots, x_m \mapsto x_{m-1}\ y_m]\ x_m = \Gamma_4 \\ \quad \mathbf{where} \\ \qquad \begin{array}{rcll} \Gamma_4 & = & \Gamma_3, & \mathbf{if}\ \rho = s \\ & = & \mathbf{U}\,\Gamma_3\ x_m, & \mathbf{otherwise} \\ \Gamma_3 & = & \mathit{update}\ \Gamma_2\ x_m\ r & \\ (r, \Gamma_2) & = & f@(\Gamma_1, y_1, \ldots, y_m) & \\ \Gamma_1 & = & \mathbf{A}\,\Gamma\ x_m & \end{array}\end{array} \tag{11}$$

Note that it is the argument of each of the applications that form the arguments to f that is passed to @, not the application itself. The function *update* takes care of the updating of a variable with a new value and is defined (*without* short-circuiting indirection chains) as:

$$\begin{array}{rcll} \mathit{update}\ \Gamma[x \mapsto v]\ x\ r & = & \mathit{update}\ \Gamma\ v\ r, & \mathbf{if}\ v \in V \\ & = & \Gamma \cup \{x \mapsto r\}, & \mathbf{otherwise} \end{array} \tag{12}$$

Note that if we are updating a variable that is an indirection to another variable, we need to update the variable that is referred to reflect the update in all variables that ultimately refer to the program that is being updated, otherwise we risk duplication of work.

Finally we need to define the $\mathbf{A}$ meta-function which evaluates each argument. This proceeds by updating each argument in turn and passing the heap obtained by each evaluation into the recursive call to $\mathbf{U}$ used to evaluate the next argument. Although the definition here implies that arguments are evaluated left to right, this ordering is arbitrary and only adopted for syntactic convenience any ordering could be adopted in practice. Arguments could even be evaluated concurrently provided appropriate care was taken.

$$\begin{array}{l}\mathbf{A}\,\Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho, x_1 \mapsto x_0\ y_1, \ldots, x_n \mapsto x_{n-1}\ y_n]\ x_{n \leq m} = \Gamma_n \\ \quad \mathbf{where} \\ \quad \begin{array}{rcll} \Gamma_i & = & \mathbf{U}\,\Gamma_{i-1}\ y_i, & \mathbf{if}\ \sigma_i = s \\ & = & \Gamma_{i-1}, & \mathbf{otherwise} \end{array}\end{array} \tag{13}$$

4 The Operational Semantics of the AAM

We represent the operational semantics as transition rules of the state of the AAM. This state is a quadruple:

$$(\mathit{Control}, \mathit{Stack}, \mathit{Heap}, \mathit{Dump})$$

where

- *Control* is a stack of instructions. These instructions are `EVAL`, `EVALARGS`, `EVALITH` and `APPLY`.
- *Stack* is a stack of variables, used as a working space to store the spine of an application when we are unwinding, with the head of the stack forming the base and the last element in the stack the root.
- *Heap* is a mapping from variables to programs as in the denotational case.
- *Dump* is a stack of *Control*-*Stack* pairs, used to hold previous states while we are working on evaluating a different part of the program.

Our machine is similar to the SECD machine [4] and the G-Machine [2, 3]. The Stack, Control and Dump used by Aladin serve similar functions as their counterparts in the SECD and G machines. The Aladin Heap is more alike the G-Machine heap, which like Aladin's heap is a global object where the graph being evaluated is held, than the SECD machine's Environment which serves a local mapping between values and variables dependant on the expression being evaluated.

The meta-function **F** evaluates a program using the state-transition rules (see below) w.r.t. a given heap (cf. `E`):

$$\begin{aligned}&\mathbf{F}\ p\ \Gamma[x \mapsto p] = \Gamma' x \\ &\quad \mathbf{where}\ (\langle\rangle, S, \Gamma', \langle\rangle) = \mathbf{T}\ (\langle \mathtt{EVAL} \rangle, \langle x \rangle, \Gamma, \langle\rangle)\end{aligned} \tag{14}$$

The meta-function **T** repeatedly applies the state-transition rules (rules 15–23 below) to a state until no more apply, returning the final state. So, if we have a transition sequence:

$$s_1 \Longrightarrow s_2 \Longrightarrow \ldots \Longrightarrow s_n$$

and no rules apply to s_n then $\mathbf{T}\ s_i = s_n, i \in 1, \ldots, n$.

We now have to give the transition rules for a state, starting from the initial state used as the argument to **T**. If none of these rules apply then the machine terminates. The first case is when the head of the stack references to a data object, that is when we have a data object applied to a number of arguments, cf. rules 1 and 8. We need to make no changes and no further work can be done in this state. If the dump is non-empty we restore it (by virtue of rule 23), else we terminate as no more rules apply.

$$\begin{array}{lcl}\langle \mathtt{EVAL} \rangle & & \langle\rangle \\ \langle x_0, x_1, \ldots, x_n \rangle & \Longrightarrow & \langle\rangle \\ \Gamma[x_0 \mapsto d \in D] & & \Gamma \\ \Delta & & \Delta\end{array} \tag{15}$$

N.B. `EVAL` can only occur in the control stack as the sole element.

If the head of the stack is an application, we need to unwind and carry on evaluating:

$$\begin{array}{lcl}\langle \mathtt{EVAL} \rangle & & \langle \mathtt{EVAL} \rangle \\ x_1 : S & \Longrightarrow & x_0 : x_1 : S \\ \Gamma[x_1 \mapsto x_0\ y_1] & & \Gamma \\ \Delta & & \Delta\end{array} \tag{16}$$

If the head of the stack refers to a function of arity m and there are less than m other elements on the stack, we have the case of a function applied to too few arguments, cf. rules 2 and 9. In this case, we need to trigger the evaluation of the strict arguments which is done using the `EVALARGS` n instruction (rule 20), where n is the number of other elements on the stack.

$$\begin{array}{lcl}\langle \mathtt{EVAL} \rangle & & \langle \mathtt{EVALARGS}\ n \rangle \\ \langle x_0, x_1, \ldots, x_{n<m} \rangle & \Longrightarrow & \langle x_0, y_1, \ldots, y_n \rangle \\ \Gamma[x_0 \mapsto f^m, x_i \mapsto x_{i-1}\ y_i] & & \Gamma \\ \Delta & & \Delta\end{array} \tag{17}$$

Note that the arguments of the function are extracted from the applications in which they reside.

If the head of the stack refers to a function of arity m and there are strictly more than m other elements on the stack, we have the case of a function applied to too many arguments, cf. rules 3 and 10. We need to evaluate the inner application which is done in a new state, after which we evaluate the result of evaluating the inner application applied to the rest of the arguments which is done by saving the control-stack pair that represents this application on the dump:

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVAL} \rangle \\ \langle x_0, x_1, \ldots, x_{n>m} \rangle & \Longrightarrow & \langle x_0, x_1, \ldots, x_m \rangle \\ \Gamma[x_0 \mapsto f^m] & & \Gamma \\ \Delta & & (\langle \texttt{EVAL} \rangle, \langle x_m, \ldots, x_n \rangle) : \Delta \end{array} \tag{18}$$

The final case for EVAL is when we have a function applied to exactly the right number of arguments (this with rule 22 reflects rules 4 and 11 in the denotational cases). We need to evaluate any strict arguments and apply the function.

$$\begin{array}{lcl} \langle \texttt{EVAL} \rangle & & \langle \texttt{EVALARGS}\ m, \texttt{APPLY} \rangle \\ \langle x_0, x_1, \ldots, x_m \rangle & \Longrightarrow & \langle x_0, y_1, \ldots, y_m, x_m \rangle \\ \Gamma[x_0 \mapsto f^m, x_i \mapsto x_{i-1}\ y_i] & & \Gamma \\ \Delta & & \Delta \end{array} \tag{19}$$

Note that as well as unpacking the arguments to the function, we keep x_m, the root of the program, as the last element of the stack. It is this variable that will be updated with the result of applying f to its arguments. In particular, if $m = 0$, that is f is a Constant Applicative Form (CAF), then it is the variable referring to the function itself that will be updated with the result.

We now need to give the state transition rules for the other instructions. First we have EVALARGS which triggers the evaluation of any strict arguments, cf. the **A** meta-function (rule 13).

$$\begin{array}{lcl} \texttt{EVALARGS}\ n : C & & e_1 \ ++ \ \ldots \ ++ \ e_n \ ++ \ C \\ x_0 : y_1 : \ldots : y_{n \le m} : S & \Longrightarrow & x_0 : y_1 : \ldots : y_n : S \\ \Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho] & & \Gamma \\ \Delta & & \Delta \\ \multicolumn{3}{l}{\quad \textbf{where}} \\ \multicolumn{3}{l}{\quad e_i \ = \ \langle \texttt{EVALITH}\ i \rangle, \quad \textbf{if } \sigma_i = s} \\ \multicolumn{3}{l}{\quad \ = \ \langle \rangle, \qquad\qquad\quad \textbf{otherwise}} \end{array} \tag{20}$$

The EVALITH i triggers the evaluation the ith element of the stack (where the head of the stack is the 0th element) in a new state:

$$\begin{array}{lcl} \texttt{EVALITH}\ i : C & & \langle \texttt{EVAL} \rangle \\ x_0 : x_1 : \ldots : x_i : S & \Longrightarrow & \langle x_i \rangle \\ \Gamma & & \Gamma \\ \Delta & & (C, x_0 : x_1 : \ldots : x_i : S) : \Delta \end{array} \tag{21}$$

By evaluating each argument in its own state we open up the possibility of evaluating all the arguments that need evaluating concurrently, a process which would be more difficult if we used the same state as the original application.

The APPLY instruction initiates the primitive application of a function to its argument, updating the root of the program and evaluating the result if necessary.

$$\begin{array}{lcl} \langle \texttt{APPLY} \rangle & & C \\ \langle x_0, y_1, \ldots, y_m, x_m \rangle & \Longrightarrow & \langle x_m \rangle \\ \Gamma[x_0 \mapsto f :: \sigma_1 \times \ldots \times \sigma_m \to \rho] & & \mathit{update}\ \Gamma'\ x_m\ r \\ \Delta & & \Delta \end{array}$$
$$\begin{array}{rcll} \multicolumn{4}{l}{\textbf{where}} \\ (r, \Gamma') & = & f@(\Gamma, y_1, \ldots, y_m) & \\ C & = & \langle\rangle, & \textbf{if } \rho = s \\ & = & \langle \texttt{EVAL} \rangle, & \textbf{otherwise} \end{array} \tag{22}$$

where the function *update* is the one defined in equation 12.

The final rule concerns the case when the control stack is empty but we have previous states on the dump. In this case, we restore the first previous state and continue. This is similar to returning from a procedure call in an imperative language.

$$\begin{array}{lcl} \langle\rangle & & C \\ S' & \Longrightarrow & S \\ \Gamma & & \Gamma \\ (C, S) : \Delta & & \Delta \end{array} \tag{23}$$

Note that we throw away the old stack, i. e., we don't in any way 'return' a value.

4.1 An Example

Suppose we have two primitive functions, *plus* and *times*, both of strictness $s \times s \to s$ giving the obvious result, and a function *square* defined such that:

$$\begin{array}{l} \mathit{square} :: s \to l \\ \mathit{square}@(\Gamma, x) = t\ x\ x \\ \quad \textbf{where } \Gamma[t \mapsto \mathit{times}] \end{array} \tag{24}$$

So, *square* returns the actual application, rather than the result of doing the application. To evaluate *square plus* 3 4, we first need an initial heap, Γ:

$$\Gamma = \{a \mapsto \mathit{times}, b \mapsto \mathit{square}, c \mapsto \mathit{plus}, d \mapsto 3, e \mapsto 4, f \mapsto c\ d, g \mapsto f\ e, h \mapsto b\ g\}$$

and then need to update this heap w.r.t. evaluating h. Our initial machine state (i. e., the one passed to **T**) is:

$$\mathit{state} = (\langle \texttt{EVAL} \rangle, \langle h \rangle, \Gamma, \langle\rangle)$$

and the transition sequence is:

$$\begin{array}{rcl} \mathit{state} & \Longrightarrow & (\langle \texttt{EVAL} \rangle, \langle b, h \rangle, \Gamma, \langle\rangle) \\ & \Longrightarrow & (\langle \texttt{EVALARGS 1}, \texttt{APPLY} \rangle, \langle b, g, h \rangle, \Gamma, \langle\rangle) \\ & \Longrightarrow & (\langle \texttt{EVALITH 1}, \texttt{APPLY} \rangle, \langle b, g, h \rangle, \Gamma, \langle\rangle) \\ & \Longrightarrow & (\langle \texttt{EVAL} \rangle, \langle g \rangle, \Gamma, \langle(\langle \texttt{APPLY} \rangle, \langle b, g, h \rangle)\rangle) \\ & \Longrightarrow & (\langle \texttt{EVAL} \rangle, \langle f, g \rangle, \Gamma, \langle(\langle \texttt{APPLY} \rangle, \langle b, g, h \rangle)\rangle) \\ & \Longrightarrow & (\langle \texttt{EVAL} \rangle, \langle c, f, g \rangle, \Gamma, \langle(\langle \texttt{APPLY} \rangle, \langle b, g, h \rangle)\rangle) \\ & \Longrightarrow & (\langle \texttt{EVALARGS 2}, \texttt{APPLY} \rangle, \langle c, d, e, g \rangle, \Gamma, \langle(\langle \texttt{APPLY} \rangle, \langle b, g, h \rangle)\rangle) \end{array}$$

$$
\begin{array}{ll}
\Longrightarrow & (\langle \texttt{EVALITH 1,EVALITH 2,APPLY}\rangle, \langle c,d,e,g\rangle, \Gamma, \langle(\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle d\rangle, \Gamma, \langle(\langle \texttt{EVALITH 2,APPLY}\rangle, \langle c,d,e,g\rangle), (\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle\rangle, \langle\rangle, \Gamma, \langle(\langle \texttt{EVALITH 2,APPLY}\rangle, \langle c,d,e,g\rangle), (\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{EVALITH 2,APPLY}\rangle, \langle c,d,e,g\rangle, \Gamma, \langle(\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle e\rangle, \Gamma, \langle(\langle \texttt{APPLY}\rangle, \langle c,d,e,g\rangle), (\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle\rangle, \langle\rangle, \Gamma, \langle(\langle \texttt{APPLY}\rangle, \langle c,d,e,g\rangle), (\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{APPLY}\rangle, \langle c,d,e,g\rangle, \Gamma, \langle(\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle\rangle, \langle g\rangle, \Gamma \cup \{g \mapsto 3+4=7\}, \langle(\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{APPLY}\rangle, \langle b,g,h\rangle, \Gamma' = \Gamma \cup \{g \mapsto 3+4=7\}, \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle h\rangle, \Gamma'' = \Gamma' \cup \{i \mapsto a\ g, h \mapsto i\ g\}, \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle i,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle a,i,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVALARGS 2,APPLY}\rangle, \langle a,g,g,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVALITH 1,EVALITH 2,APPLY}\rangle, \langle a,g,g,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle g\rangle, \Gamma'', \langle(\langle \texttt{EVALITH 2,APPLY}\rangle, \langle a,g,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle\rangle, \langle\rangle, \Gamma'', \langle(\langle \texttt{EVALITH 2,APPLY}\rangle, \langle a,g,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{EVALITH 2,APPLY}\rangle, \langle a,g,g,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle \texttt{EVAL}\rangle, \langle g\rangle, \Gamma'', \langle(\langle \texttt{APPLY}\rangle, \langle a,g,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle\rangle, \langle\rangle, \Gamma'', \langle(\langle \texttt{APPLY}\rangle, \langle a,g,g,h\rangle)\rangle) \\
\Longrightarrow & (\langle \texttt{APPLY}\rangle, \langle a,g,g,h\rangle, \Gamma'', \langle\rangle) \\
\Longrightarrow & (\langle\rangle, \langle h\rangle, \Gamma''' \cup \{h \mapsto 7 \times 7 = 49\}, \langle\rangle)
\end{array}
$$

In the final state, the variable which referred to the root of the original application now refers to 49, which is the result of *square* (*plus* 3 4).

5 An Implementation of the AAM

We choose to implement a version of the AAM, using the above semantic rules, using the programming language Java [8]. Using Java has several advantages:

- Java is platform independent.
- Java can interface with other languages, in particular C and C++, using the JNI [10].
- Java has a built-in garbage collector.
- We have a compiler for the lazy functional language Ginger [9] which generates Java class files [11, 12].

Our approach is to represent the various components of an Aladin program as Java classes.

5.1 Representation of Aladin Programs

We first need a Java representation of the four components of the AAM defined in the semantics. The Control is simply the code structure of the evaluation mechanism. The

Stack is implemented as an array which can grow on demand (we could use a Java `Stack` or `Vector` but we use an array as it enables us to perform certain operations efficiently). The Heap is a combination of the object heap of the JVM plus a hash table in which we can look up functions by their name. Finally the Dump, which in the semantics is used to simulate recursion, is implemented by actual recursion in our implementation.

The definition of programs as given in definition 5 suggests that we implement programs as a class hierarchy as seen in Figure 4. Note that some of these classes are only used by the compiler. (see Section 8). All programs are represented as sublasses of the `Prog` class:

```
public abstract class Prog {
  // ...
}
```

Variables are represented using the `Var` class:

```
public final class Var extends Prog {
  public Prog value;

  // ...
}
```

where the field `value` holds the value of the variable. Data objects are represented by instances of subclasses of the `Data` class:

```
public abstract class Data extends Prog {
  // ...
}
```

In particular, the five basic types — integers, reals, characters, booleans and strings — are represented by the classes `Int`, `Real`, `Char`, `Bool` and `Str` respectively. Functions are represented by instances of subclasses of the `Function` class:

```
public abstract class Function extends Prog {
  public String pack = "";
  public String cl = "";
  public String short_name = "";
  public int arity = -1;
  public StrictnessSig strictness;

  public final boolean strictIn(int i) {
    // ...
  }

  public final boolean hasLazyResult() {
    // ...
  }

  public abstract Prog primApply(Var[] args);
}
```

The various fields and methods are explained in Section 6. Finally applications are represented as instances of the `App class`:

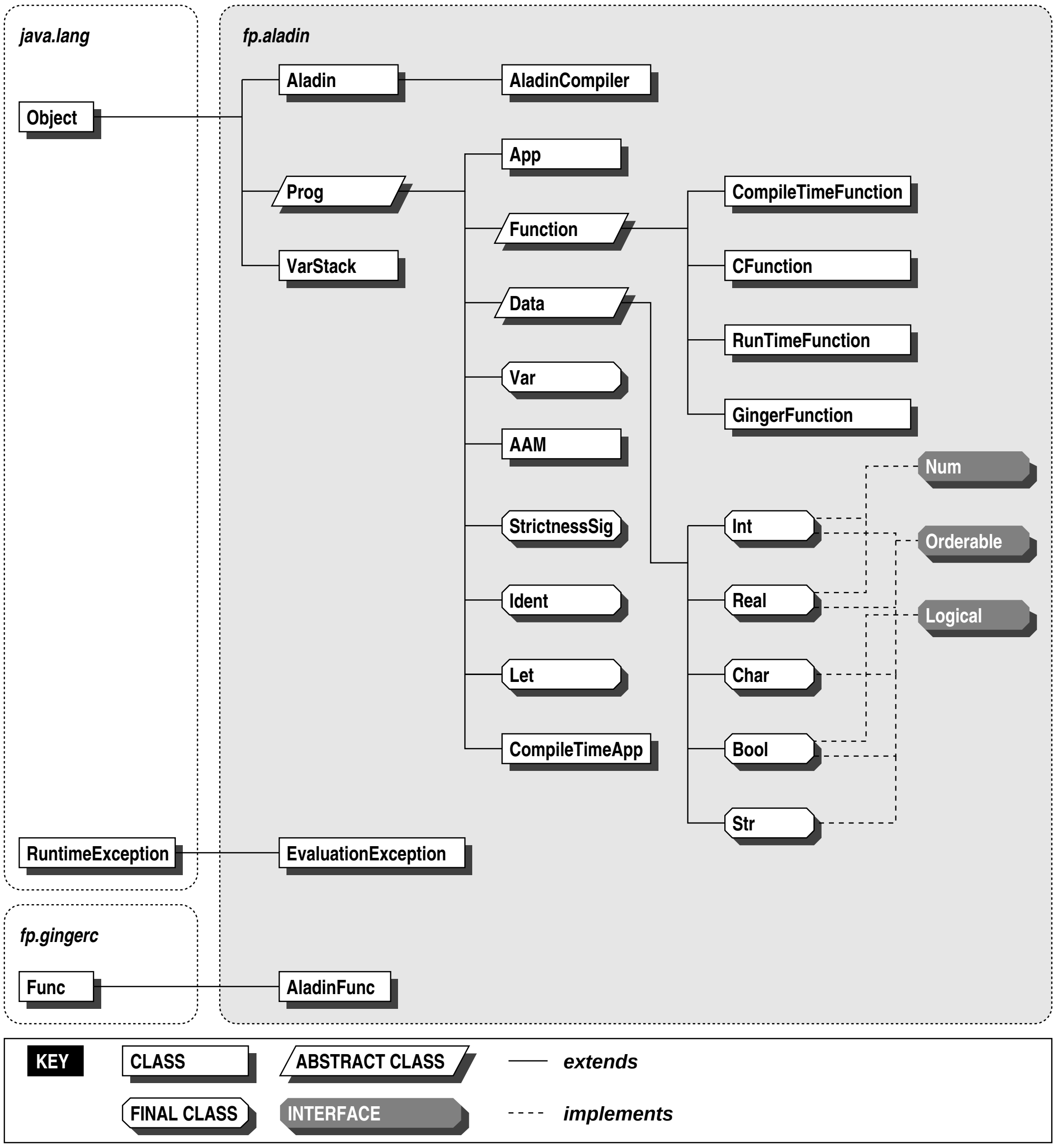

Figure 4: The `fp.aladin` package

```
public class App extends Prog {
  public Var functor;
  public Var arg;
}
```

Classes whose instances can be combined in arithmetic expressions, e. g., addition, subtraction, etc., implement the `Num` interface; classes whose instances can be ordered implement the `Orderable` interface; while classes whose instances can be combined in logical expressions, using conjunction and disjunction for example, implement the `Logical` interface.

The user is free to had his or her own types, provided that they are represented using classes which are subclasses of the `Prog` class. In the case of user-defined types, the only way to construct and deconstruct instances of them is *via* function calls: there is no special syntax such as the use of `data` and `|` constructs and pattern-matching in Haskell.

The main evaluation mechanism is contained in the `VarStack` class which is used to represent the Aladin stack. The function lookup table is held as a static field of the `AAM` class which, for convenience, will be subclassed by the classes where we define our primitives. Each of these primitives will be a represented by a static Java method (as in our Ginger compiler [11, 12]) which is reflected using the `java.lang.reflect` package as a `Method` object which is stored as a member of the `Function` class. These methods take a number of `Var` objects as arguments and return a value of type `Prog`.

5.2 The Evaluation Mechanism

The evaluation of a program is triggered by a call to the `eval` method of the `Var` object that refers to the program we wish to evaluate. This method creates a new stack and triggers the transformation of the stack (cf. the meta-function **F** defined in equation 14) is defined as follows:

```
public Prog eval() {
  (new VarStack(this)).transform();

  return get();
}
```

For convenience, the method returns the final value of the variable using the method `get` (which short-circuits chains of `Var` objects).

The method `transform` in the `VarStack` class repeatedly transforms the stack using the semantic rules 15–23 as a guide. This method starts as follows:

```
public void transform() throws EvaluationException {
  boolean eval = true;

  while (eval) {
    Var v = head();

    Object head = v.get();
```

Transformation of the stack will terminate when the `eval` flag is set to false. The method `head` returns, without popping, the `Var` at the top of the stack. If the value of this object is an `App` then we push its functor onto the stack and continue (cf. rule 16):

```
    if (head instanceof App)
      push(((App) head).functor);
```

If the value is a function we have to determine its arity. If we have too few arguments then we need to evaluate any strict ones and terminate evaluation (cf. rule 17):

```
    else if (head instanceof Function) {
      Function f = (Function) head;
      int no_args = count - 1;

      if (no_args < f.arity) {
        // rearrange the stack
        rearrange();

        // get the actual arguments and evaluate the strict ones
        eval(f, getArgs());

        eval = false;
      }
```

The method `rearrange` rearranges the stack as in rule 17 (so that we have the arguments to the application rather than the applications themselves on the stack) while the method `getArgs` copies these arguments into an array. The evaluation of the strict arguments is done by the binary method `eval` defined as:

```
private void eval(Function f, Var[] args) {
  for (int i = 0; i < args.length; i++)
     if (f.strictIn(i + 1))
        args[i].eval();
}
```

This method reflects the transformations defined in rules 20 and 21. Evaluation of the stack is now stopped by setting `eval` to false.

If we have too many many arguments for the function present, as in rule 18, we evaluate the inner application by popping the appropriate elements from the stack (the function itself and the correct number of arguments) and forming these into a new stack which we then transform. Once this new stack has been fully evaluated, we continue transforming the original one, the head of which will be the result of the evaluating the new one.

```
      else if (no_args > f.arity) {
        VarStack inner = partition(f.arity);

        inner.transform();
      }
```

The method `partition` does the job of splitting the stack up.

If we have the exact number of arguments that the function requires we can trigger the application of the function (cf. rules 19 and 22).

```
      else {
        // exactly the right number of arguments
        Var root = elements[0];

        // rearrange the stack
        rearrange();
```

```
        // discard the function and copy the args into an array
        Var[] args = getArgs();

        // evaluate the strict arguments
        eval(f, args);

        // clear the stack of all but the root
        count = 1;

        // apply the function and update the root (0th index of elements).
        Prog res = f.primApply(args);
        root.update(res);

        // the root is now the sole element of this stack
        elements[0] = root;
        count = 1;

        // keep evaluating if f has a lazy result
        eval = f.hasLazyResult();
      }
    }
```

The actual arguments of the stack are held in the array `elements` and the number of elements on the stack is stored in the variable `count`; the head of the stack is held in `elements[count - 1]` and its last element, the root, in `elements[0]`. The methods `hasLazyResult` and `primApply` are explained in Section 6.

The method `update` of `Var` deals with the updating of variable with a new value. The method looks to short-circuit any chains of variables when a variable is updated with another variable. If the value at the end of the chain cannot be further evaluated, i. e., it is a data object of a function of arity greater than zero, the the variable we are updating is updated with this value, not the variable that referred to it. This is safe as since the value cannot be further evaluated we are in no danger of replicating work.

```
public final void update(Prog p) {
  // find the end of the indirection chain
  Var v = this;
  while (v.value instanceof Var)
    v = (Var) v.value;

  if (p instanceof Var) {
    // might be able to short-circuit any chains
    Prog p_value = ((Var) p).get();

    if (p_value instanceof App)
      v.value = p_value;
    else if (p_value instanceof Function)
      v.value = (((Function) p_value).arity == 0) ? p : p_value;
    else
      v.value = p_value;
  }
```

```
  else
    v.value = p;
}
```

Note that it is the variable at the end of the indirection chain that is updated (see the text accompanying equation 12).

6 Primitives

As mentioned above, our basic mechanism for implementing primitives is the static Java method. In this section we shall only deal with how the user writes such primitives and how primitives in written in different languages are handled by Java. The mechansim of importing primitives is handled in Section 7. Since Aladin is essentially a functional language, it is expected that the user makes the primitives defined referentially transparent, that is, depend only on the value of their arguments.

Each primitive is represented by an instance of a subclass of the `Function` object. Referring back to the defintion given in Section 5.1, the fields `pack`, `cl` and `short_name` represent the package, class and short name of the function; `arity` its arity and `strictness` its strictness signature. The method `strictIn` returns whether the function is strict in a certain argument (indexing starting at 1) and `hasLazyResult` returns whether the function has a lazy result. Finally, `primApply` primitively applies the function to the given arguments

6.1 Java and Aladin Primitives

Each primitive that is written in Java (and also Aladin since we compile Aladin definitions to Java methods — see Section 8) is represented by an instance of the `RunTimeFunction` class. This has a field of type `java.lang.reflect.Method` which reflects the (static) method where the actual code is stored. To primitively apply such a function to a number of arguments merely requires applying the `invoke` method from the `Method` class to the arguments.

```
public class RunTimeFunction extends Function {
  public Method app_rule;

  public Prog primApply(Var[] args) {
    return (Prog) app_rule.invoke(null, args);
  }
}
```

Note that `primApply` only returns the result rather than the updated heap as in 6 as any updates to the heap will be done globally.

The actual writing of the Java code can be split into three parts:

1. Extract some or all of the values from the `Var` objects.

2. Do the actual work involved.

3. Wrap the result in a `Prog` object if necessary and return.

For example, consider the method `_op_plus` which is the prefix form of `+` (see Section 7.2.2).

```
public static Prog _op_plus(Var x, Var y) {
  Prog a = x.get(), b = y.get();
```

```
  if (a instanceof Num)
    return ((Num) a).plus(b);
  else
    throw new EvaluationException("Can't add " + a + " to " + b);
}
```

Note that unlike in the semantics we do not have to pass the Aladin heap as the first parameter as it already available as a static public field of the `AAM` class.

We first get the values of `x` and `y` which are placed in `a` and `b` respectively using the `get` method. This presumes that the function has been declared strict in both its arguments (see Section 7.3); if not then we can use the method `eval` in place of `get`. If it is possible to add something to `a`, i. e., `a` is an instance of a class which implements `Num`, instances of classes which implement `Num`, we call the method `plus`, else we throw an exception. The main work is done inside the `plus` method. For example, in the `Int` class, `plus` is defined as follows:

```
public Prog plus(Prog n) {
  if (n instanceof Int)
    return new Int(value + ((Int) n).value);
  else if (n instanceof Real)
    return new Real(value + ((Real) n).value);
  else
    throw new EvaluationException("Can't add " + this + " to " + n);
}
```

This method lets us add both `Int`s and `Real`s to `Int`s, giving us mixed-mode arithmetic.

As an example of a function taking a lazy argument, consider `_op_and` which is the prefix form of `&`. This has a definition similar to `_op_plus` but it is declared to have a lazy result and be lazy in its second argument:

```
public static Prog _op_and(Var x, Var y) {
  Prog a = x.get();

  if (a instanceof Logical)
    return ((Logical) a).and(y);
  else
    throw new EvaluationException("Can't take the logical conjunction of "
                                  + a + " and " + y);
}
```

Note that as the second argument is lazy, we pass `y` as a variable and don't get its value. The bulk of the work is done elsewhere; if `a` is a `Bool` we have:

```
public Prog and(Var v) {
  return (value) ? (Prog) v : (Prog) Bool.FALSE;
}
```

We see that if the first argument to`_op_and` evaluated to false we never need to evaluate the second argument (`v` in the method `and`). However, this might not always be the case. If we choose to use fuzzy logical primitives in Aladin (see, for example, [13]), where logical values are real numbers rather than booleans, then it will be necessary to evaluate the argument passed to `and`:

```
public Prog and(Var v) {
  Prog b = v.eval();

  if (b instanceof Real) {
    double v = ((Real) b).value;

    return Math.min(value, b);
  }
  else
    throw new EvaluationException("Can't take the logical conjunction of "
                                  + a + " and " + y);
}
```

6.2 C Primitives

Implementing primitives in C relies heavily on the Java Native Interface [10]. Each C function, which originates from some shared dynamic library, has a corresponding Java method declaration which is used by Aladin to interface to that function, as with the standard use of native methods in Java (see Sections 7.4.2 and 8.5 for more details). Thus, to Aladin, a C primitive is the same as a Java or Aladin one (there is a `CFunction` class for representing primitives written in C, but this is only used at compile time).

The actual code for the primitives follows the pattern for writing primitives in Java. For example, we can define the factorial function in C as follows:

```
JNIEXPORT jobject JNICALL Java_update_fac
    (JNIEnv *env, jclass cl, jobject var)
{

  /* get the value of v */
  jlong value = getInt(env, var);

  /* do the actual work */
  jlong f = 1;
  jint i;

  for (i = 2; i <= value; i++)
    f *= i;

  /* wrap the answer in an Int */
  return makeInt(env, f);
}
```

The somewhat involved-looking prototype for each function is generated by Aladin from its strictness signature using the `javah` utility and placed in a C header file. This can then be copied by cut and paste into the `.c` file where the code is to maintain consistency and save typing a complicated expression.

The first argument of the function is a pointer to an environment representing the JVM. This allows us to call Java methods, create Java objects, etc., from C (similar to the necessity of passing the heap as the first parameter of primitives in the semantics). The second argument is the class of the Java method which this function provides the implementation for; it is not used by the function. The rest of the arguments are the actual arguments of the function, in this case we have just one.

The function `getInt` gets the value of a variable which must be an integer (we are using Java longs to represent integers, hence we assign the value to a `jlong` type). This works by calling the equivalent method in Java:

```
jlong getInt(JNIEnv *env, jobject v) {
  jobject Var = getVar(env);
  jmethodID I = (*env)->GetMethodID(env, Var, "getInt", "()J");

  /* get the value of var  */
  return (*env)->CallLongMethod(env, v, I);
}
```

This code has to find the `Var` object, get an ID for its `getInt` method and finally call it. It is defined in a separate shared library (`libAladin.so` in UNIX) which must be linked in when the user compiles their code. There are similar methods — `getReal`, `getBool`, etc. — defined in both Java (as methods of the `Var` class) and in C (as global functions) as well as a general `get` function (which we used when writing primitives in Java) which returns the value of an variable as a `Prog`, which is represented as a value of type `jobject` in C. If the user needs to evaluate the variable, then they can use the functions `eval`, `evalInt`, etc.

After computing the factorial we need to wrap the result (a `jlong`) in an Aladin `Int` object. This is done using the `makeInt` function:

```
jobject makeInt(JNIEnv *env, jlong val) {
  jobject Var = getVar(env);
  jmethodID ConsInt = (*env)->GetMethodID(env, Var, "<init>", "(J)V");

  /* Invoke and return */
  return (*env)->NewObject(env, Var, ConsInt, val);
}
```

The steps take are similar to the `getInt` function. We have a whole family of functions to make Aladin programs from each primitive Java/C type. For example, `makeReal` and `makeApp` construct reals and applications respectively.

6.3 Primitives in Ginger

Each primitive written in Ginger is represented by an instance of the `GingerFunction` class:

```
public class GingerFunction extends Function {
  private fp.gingerc.Func ginger_function;
}
```

Instead of the `app_rule` field found in the `RunTimeFunction` class we have an object of the `Func` class (we explicitly qualify each class from the `fp.gingerc` package to make it clear where things are coming from and because there are some name clashes between objects) reflecting the Java method that the Ginger code has been converted to (see [12] for details on how this is done). The `primApply` method is implemented to use this field:

```
public Prog primApply(Var[] args) throws EvaluationException {
  Object[] gargs = new Object[args.length];
```

```
  for (int i = 0; i < gargs.length; i++)
    gargs[i] = args[i].toGinger();

  Object g_res = ginger_function.apply(gargs);
  g_res = fp.gingerc.Node.eval(g_res);

  return fromGinger(g_res);
}
```

Before calling the function with the given arguments, the arguments in question must be converted to a format Ginger will recognise, and similarly we must convert the result back into Aladin. Since Aladin and Ginger are both functional languages this conversion is fairly straightforward, with a couple of caveats.

6.3.1 Preventing Unnecessary Conversions

Since Ginger is lazy, it is possible that an argument, or a part of it, may pass through a call to a Ginger function completely unaltered. Converting an Aladin program to a Ginger object involves the creation of a new and distinct object, similarly with the reverse process. Suppose we have an Aladin program a which is passed to a Ginger function f; a will be converted to a Ginger function g, say. Suppose now that g is passed back as part of the result of the Ginger function (for instance as part of a list); it will be converted to an Aladin object a', say. So, a and a' are really the same object, but during the conversion process they have become divorced from each other. In particular, if we evaluate a then this evaluation is *not* reflected in a'. This can have a serious effect on performance. For instance consider the following Ginger program.

```
lists x = [x..] : lists (x + 1);

hdlists n =
    let
        ns = map hd (lists 0);
    in
        if n == 0 then
            ns
        else
            take n ns
        endif
    endlet;
```

Although `hdlists` normally runs in linear time w.r.t. the paramter `n`, if we import it into Aladin then the above conversion problem slows it down to exponential time.

The solution is to cache the Aladin-Ginger conversions so that when we wish to convert an unchanged object back from Ginger we can retrieve the original Aladin program, and *vice versa*, rather than creating a new one. This cache stores only the most recent conversions done to prevent the machine being clogged up with long irrelevant conversions and to enable Aladin and Ginger objects to be released for possible garbage collection. To minimise the conversion work done, we make sure that the Ginger object is in WHNF, by calling the `Node.eval` method.

6.3.2 Converting Aladin Functions to Ginger

Although the conversion of Aladin objects to and from Ginger is for the most part straightforward, converting Aladin functions to Ginger is slightly more complex. To solve this problem we represent Aladin functions in Ginger using the class `AladinFunc`, a subclass of the Ginger function class, `Func`.

```
public class AladinFunc extends Func {
  protected Var al_func;

  public Object apply(Object[] as) {
    Var v = al_func;

    for (int i = 0; i < as.length; i++)
      v = new Var(v, Prog.fromGinger(as[i]));

    v.eval();

    return v.toGinger();
  }
}
```

This class stores as a member the original Aladin function (or rather, a `Var` object which refers to it). When Ginger applies such a function, by calling its `apply` method, we form a new Aladin application by converting the passed arguments from Ginger to Aladin which are given as arguments to the original Aladin function. We then evaluate this function using Aladin, convert the result back to Ginger and then return.

7 A Scripting Language for Aladin

To enable us to write programs, we need a top-level language which we can program in. This language should let us import primitives from a variety of sources and combine them into programs. We choose to adopt a subset of the functional language Ginger [9] (we basically omit local function definitions) which added constructs for specifying the strictness of functions and an enhanced syntax for importing primitives. The extended BNF of this language is as follows:

$$
\begin{array}{rcl}
\langle \textit{script} \rangle & ::= & \langle \textit{packagedec} \rangle ?\ \langle \textit{decl} \rangle * \\
 & & \\
\langle \textit{packagedec} \rangle & ::= & \texttt{package}\ \langle \textit{javaid} \rangle \texttt{;} \\
 & & \\
\langle \textit{decl} \rangle & ::= & \langle \textit{def} \rangle \mid \langle \textit{import} \rangle \mid \langle \textit{id} \rangle\ \langle \textit{strictsig} \rangle \\
 & & \\
\langle \textit{import} \rangle & ::= & \langle \textit{im} \rangle\ (\langle \textit{id} \rangle\ \langle \textit{strictsig} \rangle ?)*\ \texttt{;} \\
\langle \textit{im} \rangle & ::= & \texttt{import}\ \langle \textit{class} \rangle \mid \texttt{importc}\ \texttt{"}\langle \textit{filename} \rangle\texttt{"} \mid \texttt{importg}\ \langle \textit{class} \rangle \\
 & & \\
\langle \textit{strictsig} \rangle & ::= & \texttt{::}\ \langle \textit{argstrict} \rangle ?\ \texttt{->}\ \langle \textit{strict} \rangle \\
\langle \textit{argstrict} \rangle & ::= & \langle \textit{strict} \rangle\ (\texttt{*}\ \langle \textit{strict} \rangle)*
\end{array}
$$

$$
\begin{aligned}
\langle \mathit{strict} \rangle &::= \texttt{s} \mid \texttt{l} \\[1em]
\langle \mathit{def} \rangle &::= \langle \mathit{id} \rangle\ \langle \mathit{id} \rangle * \texttt{ = } \langle \mathit{prog} \rangle \texttt{;} \\
\langle \mathit{prog} \rangle &::= \langle \mathit{let} \rangle \mid \langle \mathit{simpleprog} \rangle + \\
\langle \mathit{let} \rangle &::= \texttt{let } \langle \mathit{id} \rangle \texttt{ = } \langle \mathit{prog} \rangle \texttt{ in } \langle \mathit{prog} \rangle \texttt{ endlet} \\
\langle \mathit{simpleprog} \rangle &::= \langle \mathit{javaid} \rangle \mid \langle \mathit{data} \rangle \mid \texttt{(}\langle \mathit{prog} \rangle\texttt{)} \\
\langle \mathit{data} \rangle &::= \langle \mathit{int} \rangle \mid \langle \mathit{float} \rangle \mid \langle \mathit{bool} \rangle \mid \langle \mathit{char} \rangle \mid \langle \mathit{string} \rangle \\[1em]
\langle \mathit{javaid} \rangle &::= (\langle \mathit{id} \rangle\ \texttt{.}) *\ \langle \mathit{id} \rangle \\
\langle \mathit{package} \rangle &::= \langle \mathit{javaid} \rangle \\
\langle \mathit{class} \rangle &::= (\langle \mathit{package} \rangle\ \texttt{.})?\ \langle \mathit{id} \rangle
\end{aligned}
$$

where *id* is any valid Java identifier containing *no* period ('`.`') separators, and *filename* is a legal file name, the syntax of which is dependent on the operating system we are running under. An Aladin script is thus an optional package declaration followed by a number of declarations. These declarations are:

Import declarations used to import primitives from a class.

Strictness declarations used to specify the strictness of a primitive.

Definitions used to specify a primitive function by giving a definition which constructs a graph from its arguments and other primitives.

Each script file will be appened with the suffix '`.as`'. As with our Ginger compiler [11, 12], we compile scripts written in this language into a Java class file, translating each definition into a static Java method, and generating Java code to deal with the importing of functions and the setting of their strictnesses.

7.1 Package Declarations

A package declaration, which must be the first statement of an Aladin script, follows the same syntax as package declarations in Java and has the same meaning. All primitives defined in this file will be placed inside a class with will be in the package declared by the user. If no package declaration is made then then empty package is used. For instance the declaration:

```
package fp.aladin;
```

places all the following definitions in the package `fp.aladin`.

7.2 Definitions

Aladin definitions follow the normal syntactic style found in functional languages, in particular the style of Ginger. The five basic data types — integers, reals, strings, characters and booleans — follow the normal syntax. Applications are written using juxtaposition (and associate to the left) and we allow the use of infix operators, standard functional list notation (see below) and simple local variable definitions *via* the let declaration. We also use the Ginger `if-then-elsif-else-endif` notation for conditionals (again see below for more details). For example, the Fibonacci function can be defined in Aladin as:

```
fib :: s -> l
fib n =
    if n == 0 then 1
    else
        let
            f1 = fib (n - 1);
            f2 = fib (n - 2);
        in
            f1 + f2;
        endlet
    endif;
```

See Section 7.3 for more details on strictness signatures.

This definition merely takes its arguments and produces the graph described in its left-hand side, hence we have to return a lazy result as we want the graph to be evaluated. No evaluation or compiler optimisations are performed, these being handled by the evaluation mechanism defined in the previous section: the construction and evaluation of programs are completely divorced in the scripting language.

7.2.1 Function Names

Since our programs are ultimately represented by Java classes, it follows that function names in Aladin follow the same syntax as they do in Java. In particular a function name has three parts: its short name, its class, and its package. We may refer to a function either by its short name, its class plus its short name (separated by a period), or its package plus its class plus its short name (again separated by periods). For example, the function `foo` in the class `Bar` in the package `fp.aladin` can be referred to as either `foo`, `Bar.foo` or `fp.aladin.Bar.foo`. Of course, functions in different classes may have the same short name; to differentiate them we adopt the policy that we choose the last defined/imported one. For example, if `foo` is also defined in the class `Doe` which is imported after the `foo` in `Bar` then the short name `foo` refers to the one in `Doe` and not in `Bar`. If we wish to use the version in `Bar` then we can always qualify the name further by writing `Bar.foo`. This has the obvious extension to functions and classes with the same names but different packages.

7.2.2 Infix Operators

We allow the use of infix operators, though these are converted to prefix functions during the parsing stage according to the precedence rules defined in Table 1 (we include the already prefix not, `~` for completion) . Note that in the case of '`.`', `foo . bar` is a function composition but `foo.bar` is a single identifier. Also, `-` is used for both binary minus and to indicate a negative *constant*, but not for general unary minus, i. e., `-3` is legal but `-x` isn't.

Some of these operators are overloaded, the resolution being done at runtime. In these cases, the prefix form only exists to unwrap the arguments and call the interface method. Any user-defined class which implements the method specified in the fourth column adds another overloading to the prefix form. For instance, if we have a class `Foo` which implements the `Orderable` interface then we can use the comparison operators on objects of that type.

Prec.	Infix form	Prefix form	Method
0	`++`	`_op_list_cat`	N/A
	`:`	`_op_list_cons`	N/A
1	`|`	`_op_or`	`Logical.or`
2	`&`	`_op_and`	`Logical.and`
3	`==`	`_op_eq`	`Object.equals`
	`~=`	`_op_ne`	`Object.equals`
	`<`	`_op_lt`	`Orderable.lt`
	`<=`	`_op_le`	`Orderable.le`
	`>`	`_op_gt`	`Orderable.gt`
	`>=`	`_op_ge`	`Orderable.ge`
4	`+`	`_op_plus`	`Num.plus`
	`-`	`_op_minus`	`Num.minus`
5	`*`	`_op_times`	`Num.times`
	`/`	`_op_divide`	`Num.div`
	`%`	`_op_modulus`	`Num.mod`
6	`^`	`_op_exp`	`Num.exp`
7	`.`	`_op_compose`	N/A
8	`!`	`_op_list_index`	N/A
9	`~`	`_op_not`	`Logical.not`

Table 1: Aladin infix operators and their prefix form

7.2.3 Conditionals

As with infix operators, the Ginger style `if` syntax, which is adopted to reduce excessive bracketing, is converted to a prefix function application. The expression:

`if` a_1 `then` c_1 `elsif` a_2 `then` $c_2 \ldots$ `elsif` a_n `then` c_n `else` d `endif`

is converted into the program:

`if` a_1 c_1 (`if` a_2 c_2 $(\ldots($`if` a_n c_n $d)\ldots))$

Where the latter `if` is a three-argument function which returns its second argument if its first evaluates to true and its third otherwise.

7.2.4 Lists

We also allow the use of the standard square bracket to denote lists, with the list

`[`x_1 `,` x_2 `,` $\ldots$ `,` x_n`]`

being translated into multiple applications of the cons function:

x_1 `:` x_2 `:` $\ldots$ `:` x_n `:` `[]`

with the `:` operators being further converted to its prefix form. We also allow the use of the 'dot-dot' notation, again converting to a prefix function application defined in Table 2.

The prefix functions corresponding to the infix operators are *not* automatically imported by Aladin and the user is free to provide whatever definitions they see fit, though we do provide a simple implementation which the user is free to import. Similarly, there is no fixed implementation of lists, allowing the user to provide their own, though again we provide a simple implementation.

Dot-dot form	Prefix form
`[]`	`_op_list_empty`
`[` a `.. ]`	`from` a
`[` a `..` b `]`	`fromTo` a b
`[` a `,` b `.. ]`	`fromThen` a b
`[` a `,` b `..` c `]`	`fromThenTo` a b c

Table 2: Aladin 'dot-dot' lists and their prefix form

7.3 Strictness Signatures

A strictness declaration declares the strictness of a primitive function's arguments and result. If the primitive in question is defined in the script, then this signature can appear anywhere in the script but *must* be given; if the primitive is imported then a strictness signature can be given in the import declaration (see Section 7.4) in which case it overrides the original signature (if there was one).

An argument or result is either strict, denoted by an `s`, or lazy, denoted by an `l`. The arguments of a primitive defined in a strictness signature are separated using the symbol `*`, and the arguments are separated from the result by the symbol `->`, which is present even if the primitive in question takes no arguments. It is an error to give a strictness signature to a non-function, or to a function which is neither imported nor has no definition. As an example, we can give the strictness of the primitives imported from the class `List` (see Section 7.4):

```
import fp.aladin.lib.List
    _op_list_cons   :: l * l -> s
    _op_list_empty  :: -> s
    _op_list_index  :: s * s -> s
    _op_list_length :: s -> s
    isEmpty         :: s -> s
    hd              :: s -> l
    tl              :: s -> l
```

Thus `_op_list_cons` (the prefix form of `:`, see above) takes two lazy arguments and returns a strict result; `_op_list_empty` returns a strict result; `_op_list_index` takes two strict arguments and returns a strict result; `_op_list_length` and `isEmpty` take a strict result and return a strict result; and finally `hd` and `tl` take a strict argument and return a lazy result.

The number of arguments referred to in a strictness signature refer to the number of *explicit* arguments given, not to any implied by η-conversion. For instance, if we define:

```
hd1 = hd;
```

then the strictness signature of `hd1` is `-> s` (since it takes no arguments and returns a function) and not that of `hd` (`s -> l`).

7.4 Import Declarations

Each import declaration, of which there might be any number and in any order, is used to import primitives from a specified language and source. The languages currently supported are Aladin itself, Java, Ginger and C/C++. Primitives defined in the foremost two languages are imported using the `import` declaration; primitives from Ginger are imported using the `importg` declaration; and primitives from C/C++ by the `importc` declaration.

7.4.1 Importing Aladin and Java Primitives

As we shall see in Section 8, we shall compile Aladin definitions into static Java methods, so the import procedure for primitives defined in Aladin and Java is the same. The import declarations are written as part of the static initialiser of the Java class generated by the Aladin compiler (see Section 8.4) and hence the when the class is first loaded it will trigger the import procedure. In particular, any imports done in the class we are importing will be done first.

The first code inside the static initialiser creates the function, setting its strictness signature and inserting it into the heap, but for the moment leaving its `app_rule` (see Section 6.1) null for the moment. As an example, we have the following from `stdlib.java` the code generated from `stdlib.as` where the arithmetic functions, amongst others, are defined:

```
static {
  put("fp.aladin.lib", "Operators", "_op_plus", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_minus", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_times", CONST_0);
  put("fp.aladin.lib", "Operators", "_op_divide", CONST_0);

  // ...
}
```

The static `put` method (inherited from the `AAM` class) creates a new function from its arguments and inserts it into the heap. The first three arguments represent the package, class and name of the function to be created. The last argument is the strictness signature, which is created only once and stored as a private field of the generated class (see Section 8.2). Here `CONST_0` represents the strictness signature $s \times s \to s$.

After setting the functions up, we have to fill in their `app_rule` field. This is done by going through each class, including the class that is doing its importing since any functions it defines itself are held in that class, using the `getDeclaredMethods` method from the `java.lang.Class` class and filling in each function with its appropriate `Method` object. If we end up with any functions which we have a strictness signature for but no corresponding `Method` then an error has occurred.

7.4.2 Importing C Primitives

Each primitive that is written in C and directly imported will have had a corresponding Java method header generated in the generated class file, hence these primitives will be imported by the same mechanism that imports Java and Aladin primitives. However, we also need to load in the libraries where the actual object code can be found. This is done by loading each library using the `System.loadLibrary` method.

7.4.3 Importing Ginger Primitives

This is done similarly to importing Aladin and Java primitives, except that as each Ginger function is stored as a `fp.gingerc.Func` field of the Ginger class generated by the Ginger compiler from each Ginger script we have to examine the fields of each Ginger class rather than the methods.

8 Compilation

The aim of the compiler is to take an aladin script and translate it into a Java program which will then be compiled by a Java compiler into a Java class file. Our compiler has three jobs:

1. To provide a static Java method for each primitive defined in the script.
2. To set the strictness of each function defined in the script and any imported function whose strictness is redefined in the script.
3. To import all the functions specified in the script.

The latter two jobs will be accomplished by putting the code that achieves the object in a static initialiser of the created class (see above). This means that when the class is first referenced (usually by an `import` declaration) the first thing that it will do is import the functions it needs and set their strictnesses.

8.1 Compiling Scripts

To compile a script we need the following parameters:

- c, the class to create (derived from the name of the script).
- p, the package to place the created class in, declared using `package` (if no such declaration the p is set to the empty string).
- ss, the set of *distinct* strictness signatures used in a script.
- cs, the set of distinct constants used in a script.
- fs, the set of functions defined in the script with those imported into the script who have their strictness signature set or over-ridden.
- cls, the classes containing primitives defined in Java and Aladin *directly* imported into a script.
- gs, the classes containing primitives defined in Ginger directly imported into a script.
- ls, the shared object libraries containing primitives defined in C/C++ directly imported into a script.
- ds, the functions defined in the script (with their definitions) plus any primitives defined in C which are directly imported.

The compilation scheme $\mathcal{P}$ creates a Java program using these parameters

$$
\begin{array}{lcl}
\mathcal{P}\ p\ c\ ss\ cs\ fs\ cls\ gs\ ls\ ds & = & \texttt{package}\ p\texttt{;} \\
 & & \texttt{import fp.aladin.*;} \\
 & & \texttt{public final class}\ c\ \texttt{extends AAM}\ \{ \\
 & & \quad \mathcal{S}\ ss\ cs\ fs\ cls\ gs\ ls\ ds \\
 & & \}
\end{array}
$$

If no package has been given then we omit the `package` declaration. The $\mathcal{S}$ scheme creates the static initialiser for the created class which deals with setting the strictnesses of the function defined and imported and importing the necessary classes and libraries (see the next section).

8.2 Compiling Constants and Strictness Signatures

We do not create a separate object representing all the constants and strictness signatures in a class. Rather for each distinct constant we create just one instance and store it as a private field of the class we are creating and read this field whenever we want an instance of the particular constant or strictness signature.

$\mathcal{S}\ (s_1, \ldots, s_m)\ (c_1, \ldots, c_n)\ fs\ cls\ gs\ ls\ ds$

$=$ `static {`

$\quad cc\ 1\ c_1$

$\quad \ldots$

$\quad cc\ n\ c_n$

$\quad cs\ (n+1)\ s_1$

$\quad \ldots$

$\quad cs\ (n+m)\ s_m$

$\mathcal{D} fs\ cls\ gs\ ls\ ds\ \rho$

where

$$\rho = \left[\begin{array}{l} c_1 \mapsto \texttt{CONST\_}i, \ldots, c_n \mapsto \texttt{CONST\_}n, \\ s_1 \mapsto \texttt{CONST\_}(n+1), \ldots, s_m \mapsto \texttt{CONST\_}(n+m) \end{array} \right]$$

The function cc compiles a constant and places it in a static private field of the class being created:

$cc\ i\ c =$ `private static Var CONST_`i $=$ `new Var(`c`);`

The `Var` class has a constructors to construct variables which point to the appropriate program for each basic type. The function cs is similar to cc but it compiles a strictness signature instead.

$cs\ i\ (\sigma_1 \times \ldots \times \sigma_n \rightarrow \rho)$

$=$ `private static StrictnessSig CONST_`i $=$

$\quad$ `new StrictnessSig(`as`, `r`);`

where

as	$=$	`new boolean[] {`a_1`, ..., `a_n`}`	
a_i	$=$	`true`,	**if** $\sigma_i = s$
	$=$	`false`,	**otherwise**
r	$=$	`true`,	**if** $\rho = s$
	$=$	`false`,	**otherwise**

8.3 Compiling Strictness Declarations

The $\mathcal{D}$ scheme declares the strictness signature of each function. The final parameter is an environment detailing which constant or strictness signature corresponds to each field. For a constant or strictness signature, c its corresponding field is $\rho(c)$.

$\mathcal{D}\ (f_1, \ldots, f_m)\ cls\ gs\ ls\ ds\ \rho$

$=$ `put(`p_1`, `c_1`, `n_1`, `s_1`);`

$\quad \ldots$

$\quad$ `put(`p_m`, `c_m`, `n_m`, `s_m`);`

$\mathcal{I}\,cls\ gs\ ls\ ds\ \rho$

where

$$\begin{array}{rcl} p_i &=& package(f_i) \\ c_i &=& class(f_i) \\ n_i &=& name(f_i) \\ s_i &=& \rho(strictness(f_i)) \end{array}$$

8.4 Compiling Imports

The $\mathcal{I}$ scheme deals with creating the code needed to import the various classes and libraries.

$\mathcal{I}(c_1, \ldots, c_n)\ (g_1, \ldots, g_m)\ (l_1, \ldots, l_k)\ ds\ \rho\ =$

```
  importClass("c_1");
  ...
  importClass("c_n");
  importGinger("g_1");
  ...
  importGinger("g_m");
  System.loadLibrary("l_1");
  ...
  System.loadLibrary("l_k");
}
```

$\mathcal{H}\ ds\ \rho$

Note that this scheme includes the brace which terminates the static initialiser started by $\mathcal{S}$.

8.5 Compiling Definitions

The $\mathcal{H}$ compiles all the definitions in a script plus creates headers for any primitives written in C into the script:

$$\begin{array}{rcl} \mathcal{H}(f_1, \ldots, f_n)\rho &=& \mathcal{F}\ f_1\ \rho \\ && \ldots \\ && \mathcal{F}\ f_n\ \rho \end{array}$$

The scheme $\mathcal{F}$ compiles an individual function. If the function is written in C we just need to declare a native method:

$\mathcal{F}\ f^n\ \rho =$ `public final native static Prog` f`(Var x_1, ..., Var x_`n`);`

where `x_`i are dummy parameter names and n is the arity of the function. Each definition first has all its variables renamed so that they are all distinct and then compiled using $\mathcal{F}$: $\mathcal{F}$ scheme:

$\mathcal{F}\ (f\ x_1 \ldots x_n$ `=` $E)\ \rho$

```
  = public final static Prog f(Var x_1, ..., Var x_n) {
      Var v_1 = C D_1 ρ vs;
      ...
      Var v_m = C D_m ρ vs;
```

```
  return R E' ρ vs;
}
where
  (⟨(v_1, D_1), ... (v_m, D_m)⟩, E') = fl E
  vs = {x_1, ..., x_n, v_1, ..., v_m}
```

To simplify matters, all local variables are 'floated' to the top level using the *fl* function. This function splits a program into a list of declarations and a program containing no `let`s:

$$\begin{array}{rcl}
\mathit{fl}\ ((\texttt{let}\ v\ \texttt{=}\ D\ \texttt{in}\ B\ \texttt{endlet})\ C) & = & ((v, D') : ds\ +\!\!+\ es, P) \\
 & & \textbf{where} \\
 & & \quad \begin{array}{rcl} (ds, P) & = & \mathit{fl}\ (B\ C) \\ (es, D') & = & \mathit{fl}\ D \end{array} \\
\mathit{fl}\ (C\ (\texttt{let}\ v\ \texttt{=}\ D\ \texttt{in}\ B\ \texttt{endlet})) & = & ((v, D') : ds\ +\!\!+\ es, P) \\
 & & \textbf{where} \\
 & & \quad \begin{array}{rcl} (ds, P) & = & \mathit{fl}\ (C\ B) \\ (es, D') & = & \mathit{fl}\ D \end{array} \\
\mathit{fl}\ (BC) & = & (ds\ +\!\!+\ es, B'\ C') \\
 & & \textbf{where} \\
 & & \quad \begin{array}{rcl} (ds, B') & = & \mathit{fl}\ B \\ (es, C') & = & \mathit{fl}\ C \end{array} \\
\mathit{fl}\ P & = & (\langle\rangle, P)
\end{array}$$

In a conventional compiler we would have to be careful about how far outwards we floated local variable declarations as we might end up unnecessarily building the graph of programs that could otherwise be avoided. For instance, in a conditional expression a conventional compiler can exploit its knowledge of the conditional primitive to build only the graph related to the 'true' branch of the conditional and ignoring all the rest. Since Aladin knows very little about how its primitives work we can perform no such optimisation, but this does mean we don't have to be as careful about where we place our local variable declarations as in a conventional compiler.

The $\mathcal{R}$ and $\mathcal{C}$ scheme compile a simple program. They differ only in how they treat the outermost part of a program when that part is an application. If we have a constant then we simply have to load the relevant field:

$$\mathcal{C}\ c\ \rho\ vs = \mathcal{R}\ c\ \rho = \rho(c)$$

If we have an identifier then we either have a function or a variable. We can tell the difference by seeing if the identifier is in the set of variables passed to $\mathcal{C}$:

$$\begin{array}{rcll}
\mathcal{C}\ id\ \rho\ vs & = & id, & \textbf{if}\ id \in vs \\
 & = & \texttt{get(}id\texttt{)}, & \textbf{otherwise}
\end{array}$$

If we have a variable then the code to compile is thus that variable, otherwise we presume it is a function name and look it up in the heap using the `get` function inherited from the `AAM` superclass.

If we have an application, then the $\mathcal{C}$ scheme compiles the functor and the argument using the $\mathcal{C}$ scheme and then forms then into an `App` which is pointed to by a `Var` (achieved using the two-argument constructor of `Var`):

$$\mathcal{C}\ (f\ a)\ \rho\ vs = \texttt{new Var(}\mathcal{C}\ f\ \rho\ vs\texttt{,}\ \mathcal{C}\ a\ \rho\ vs\texttt{)}$$

If we are compiling the outermost application then we do not need to create the `Var`. Hence $\mathcal{R}$ is defined as:

$$\mathcal{R}\ (f\ a)\ \rho\ vs = \texttt{new App(}\mathcal{C}\ f\ \rho\ vs\texttt{,}\ \ \mathcal{C}\ a\ \rho\ vs\texttt{)}$$

8.6 Compiling the Target Code and Using the Resultant Classes

The code created by the compiler is placed in a `.java` file which is compiled into a class file using the `javac` compiler. If the script directly imports any C/C++ functions then `javah` is run over the generated class to create the header file defining the prototypes for the imported C/C++ functions.

Each class inherits a `main` method from its `AAM` superclass. When the class is executed, using the `java` interpreter for example, this method forms any arguments present into a single string which it passes through the parser to obtain a simple program (i. e., one with no local variable declarations) which it then evaluates. If no arguments are present then an attempt is made to find an evaluate a CAF named `main`. For example, suppose we have the following example in the script `foo.as`:

```
import fp.aladin.stdlib;
import fp.aladin.gstdlib;

importg bar
    lists :: s -> l
    hdlists :: s -> l;

main = hdlists 10;
```

where `lists` and `hdlists` are as defined in Section 6.3. Then this script is compiled into the Java file `foo.java`:

```
import fp.aladin.*;

public class foo extends AAM {
  private static fp.aladin.Var CONST_2 = new fp.aladin.Var(10);
  private static fp.aladin.StrictnessSig CONST_0 =
    new fp.aladin.StrictnessSig(new boolean[] {true}, false);
  private static fp.aladin.StrictnessSig CONST_1 =
    new fp.aladin.StrictnessSig(new boolean[] {}, false);

  static {
    putGinger("", "bar", "lists", CONST_0);
    putGinger("", "bar", "hdlists", CONST_0);
    put("", "foo", "main", CONST_1);

    importClass("fp.aladin.stdlib");
    importClass("fp.aladin.gstdlib");
    importClass("foo");
    importGinger("bar");
    System.loadLibrary("Aladin");
  }
```

```
  public static Prog main() {
    return new fp.aladin.App(get("hdlists"), CONST_2);
  }
}
```

This is then compiled into the class `foo.class`. The class `fp.aladin.stdlib` contains various standard functions, while `fp.aladin.gstdlib` contains import declarations and strictness signatures for Ginger primitives and functions in its standard prelude.

We can now evaluate programs. To evaluate `main` we simply pass `foo` as an argument to `java` with no extra arguments:

```
gem:~/Aladin/progs> java foo
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can also apply `hdlists` to a different argument:

```
gem:~/Aladin/progs> java foo hdlists 20
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Since we import `stdlib` into our script, we have access to standard functions such as arithmetic and comparison operators, and hence we can evaluate programs using these functions *via* the `foo` class:

```
gem:~/Aladin/progs> java foo '(3 < 2) | (2 >= 8 + 5)'
false
```

9 Conclusion and Further Work

We have given the denotational and operational semantics of the Aladin Abstract Machine and an implementation of the machine, written in Java and creating a Java class. These semantics detail a very simple functional language, where each primitive can be written in any of Java, C/C++, Ginger or the Aladin scripting language that we developed, and has the strictness of its arguments and result specified by the user.

The AAM presents us with many possible avenues of investigation. Of particular concern to us is the investigation of the effect of strictness on the space/time performance of programs and the use of partial and parallel implementation methods. The use of Aladin is obviously of use for the former; with the latter, knowing the strictness of functions means we know which parts of a program can be evaluated and hence which parts can be evaluated in parallel or partially. The simplicity of the machine also saves us from unnecessary complications that arise from pattern matching, conditionals, etc.

Other possible future projects using the Aladin model include extending the range of source languages covered, investing the use of type systems with Aladin, looking at the use of logic languages (for example, Prolog) with Aladin, and the integration of Aladin with Object technologies such as CORBA (Common Object Request Broker Architecture) [14].

References

[1] T. Axford and M. Joy, "Aladin: An Abstract Machine for Integrating Functional and Procedural Programming," *Journal of Programming Languages*, vol. 4, pp. 63–76, 1996.

[2] T. Johnsson, "Efficient Compilation of Lazy Evaluation," in *Proceedings of the 11th ACM Symposium of Principles of Programming Languages*, pp. 58–69, 1984.

[3] S. L. Peyton Jones, *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[4] P. Landin, "The Mechanical Evaluation of Expressions," *BCS Computing Journal*, vol. 6, pp. 308–320, January 1964.

[5] J. Fairbairn and S. Wray, "TIM: A simple, lazy abstract machine to execute supercombinators," in *Functional Programming Langauges and Computer Architecture*, no. 274 in LNCS, pp. 34–45, Springer-Verlag, 1987.

[6] B. Kernighan and D. Ritchie, *The C Programming Language.* Prentice Hall, 2nd ed., 1988.

[7] B. Stroustrup, *The C++ Programming Language.* Addison-Wesley, 3rd ed., 1997.

[8] K. Arnold and J. Gosling, *The Java Programming Language.* Addison-Wesley, 2nd ed., 1998.

[9] M. Joy, "Ginger — A Simple Functional Language," Tech. Rep. CS-RR-235, Department of Computer Science, University of Warwick, Coventry, UK, 1992.

[10] R. Gordon, *Essential JNI: Java Native Interface.* Prentice Hall, 1998.

[11] G. Meehan, "Compiling functional programs to Java byte-code," Research Report CS-RR-334, Department of Computer Science, University of Warwick, Coventry, UK, September 1997.

[12] G. Meehan and M. Joy, "Compiling Lazy Functional Programs to Java Byte-code." Submitted to *Software — Practice and Experience*, 1998.

[13] G. Meehan and M. Joy, "Animated Fuzzy Logic," *Journal of Functional Programming*, vol. 8, November 1998.

[14] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communcations Magazine*, vol. 35, February 1997.