THE UNIVERSITY OF CALGARY

The CHARM Project: A Back End to the Charity Interpreter

by

Barry Yee

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

COURSE OF CPSC 502

Instructor M. Williams

Supervisor R. Cockett

CALGARY, ALBERTA

Abstract

The CHARM project is an attempt to build an efficient and portable abstract machine for the Charity language. The abstract machine is implemented in C programming language to make the system more portable across different computer platforms. In this paper, the Charity system is composed of a set of rewrite rules that are converted to state transitions that defines an abstract machine. The implementation of the system is divided into two levels, the macro–level and micro–level. Macro–instructions are instructions that specify the state transitions while the micro–instructions perform the actions that change the state of the machine.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank a few individuals who have contributed to the CHARM project. To members of the Charity research group Dr. Robin Cockett for his support and guidance throughout this project and Tom Fukushima for the help in understanding Charity. In addition to Dr. Mike Williams for his instruction in research methodologies and finally to Marc Schroeder for his suggestions and related work on the Charity interpreter (CHIRP).

1 Background

1.1 Charity Basics

The Charity project is an attempt at developing a complete interpreter for the Charity language[CF92]. The Charity project is composed of two components, the front end – user interface (CHIRP) and the back end – abstract machine (CHARM). The task of CHIRP is to take programs specified in the Charity language (term logic) and translate it into *categorical combinators*. Charity programs represented as categorical combinators are then evaluated by the CHARM abstract machine. Figure 1 gives a breakdown of the Charity project and the interface between its two parts. The focus of this paper is on the CHARM aspect of the Charity project.

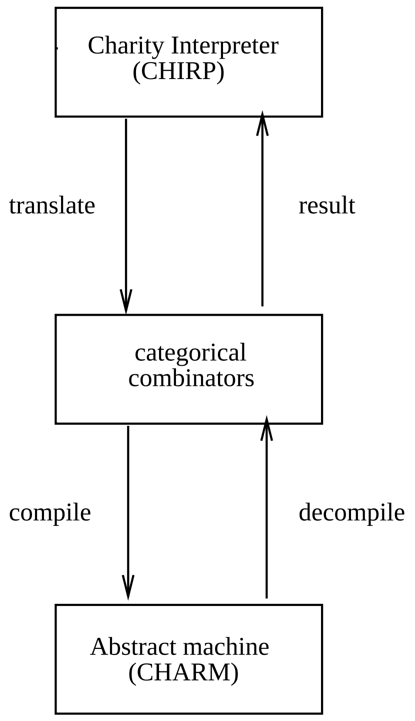

Figure 1: CHIRP and CHARM interface

Charity programs are not directly executed, they are translated into categorical combinators to provide an intermediate interface between the front end (term logic) and the back end (abstract machine). A sequence of categorical combinators composed together make up a categorical combinator expression and define a Charity function.

While it is possible to program at the combinator level this level of programming is analogous to assembly language programming.

For example the following combinator expression defines multiplication:

$$\langle P_1; \mathrm{id}, \mathrm{id}\rangle; \mathrm{fold\_nat}(\langle P_1; P_0; \mathrm{id}, !; \mathrm{zero}\rangle, \langle P_0; P_0; \mathrm{id}, \langle P_0; P_0; \mathrm{id}, P_0; P_1; \mathrm{id}\rangle; \mathrm{add}\rangle); P_1$$

It is not immediately obvious that the above combinator expression is the defines the multiplication function over the natural numbers. In Charity term logic, the multiplication function defined below is considerably more readable[1]:

```
def mul (x, y) =
    p1
      ({| zero:()     => (x,zero)
        | succ:(x, y) => (x,add(x, y))
        |} (y)
      ).
```

A rewriting system then specifies how the categorical combinator expressions are evaluated. Implementing a rewriting system is done by an abstract machine

[1]This multiplication function can be found in the EXAMPLES directory of the SML implementation of the Charity system

that can be thought of as a state transition machine with behavior governed by a set of rewrite rules. State transitions in the abstract machine are derived from the the rewrite rules. The objective in designing an efficient abstract machine lies in selecting state transition rules that are as mechanical and simple as possible thereby minimizing the complexity of the machine. At the same time the state transition rules should translate easily from the source language to the abstract machine level[Pey].

1.2 Previous Work

The Charity system described in [CF92] uses an "eager" or "by–value" abstract machine to evaluate programs. The eager abstract machine is a high level – evaluation model implemented in the SML programming language and modeled after the *Categorical Abstract Machine* (CAM) [CCM85].

The CAM machine is said to be "eager" because all arguments to functions are evaluated fully before the function is evaluated. The choice of an eager evaluation strategy has potentially serious efficiency problems since unnecessary computation is performed. In [Her92] some of the efficiency problems of the eager Charity machine are addressed using a "lazy" evaluation model. Hermann's model, also called CHARM suspends evaluation of all arguments to functions until they are required by the function. In addition, Hermann's CHARM machine introduces several graph reduction techniques to improve the efficiency of the abstract machine (see Appendix A).

1.3 Goal and Achievements

The motivation behind this project is to build a portable and reasonably efficient abstract machine that will execute Charity programs quickly. The abstract machine described in this paper is implemented in the C programming language so as facilitate the portability of the system.

The first step in this project was to re–implement the by–value machine in the C programming language. The C implementation removes most of the pattern matching associated with the SML implementation. In addition, functions defined as categorical combinator expressions are compiled into “machine code”. In this paper the term machine code refers to two levels, the macro–code and micro–code. The macro–code is a lower level specification of categorical combinators where each macro–instruction is comprised of micro–instructions. At a micro–code level, micro–instructions change the state of the machine. Figure 2 illustrates the layers of the CHARM abstract machine.

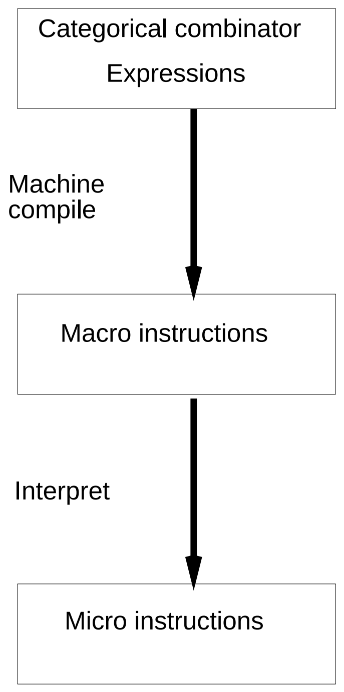

Figure 2: Overview of the CHARM abstract machine

The layout of this paper is as follows. Section 2 introduces the rewriting system of Charity that serves as a foundation for the remainder of the paper. From the rewriting rules, an abstract machine is derived and presented in Section 3. In

Section 4 (The Macro–code) and Section 5 (The Micro–code) a description of a C implementation of the eager abstract machine is given. Finally in Section 6, a suggestion of future work is discussed.

2 The Rewriting Rules of Charity

Charity programs are accepted by the CHIRP user interface and translated to categorical combinator expressions by the translation rules described in [CF92]. Categorical combinator expressions satisfy the basic rules of category theory that makes provision for an identity combinator and allows them to be composed together with an associative composition. In addition, basic category theory contains objects and maps (arrows) between objects. In categorical combinator terms, the objects are the datatypes and a map from one datatype to another is defined by giving a composition of combinators.

Evaluating categorical combinator expressions are done through a rewriting system. In a rewriting system, expressions are replaced by a "reduction" with an equivalent expression. For the discussion in this paper, the terminology "reduction" will mean a single step in the evaluation of a Charity program. Combinator expressions are executed through a series of reductions to transform one combinator expression to another, until no more reductions can be performed. An expression where no more reductions can be performed is said to be in reduced or normal form.

A reduction in a rewriting system is determined by a set of rewrite rules. The rewrite rules described in this section use the following notation:

$\langle a, b\rangle$	– represents a pair
$;$	– is the symbol for composition
$\Rightarrow$	– is a rewrite
$\vert$	– is a choice operator

Categorical combinator expressions are represented in a graph structure where a program composed of combinators is evaluated by reducing this graph structure using a set of rewrite rules. Each individual categorical combinator can be thought of as a function with parameters. For example, the pair combinator is denoted by $\langle a, b\rangle$ and has an alternative infix notation PAIR(a, b); when $\langle a, b\rangle$ is composed with the first projection P_0 it is written as $\langle a, b\rangle; P_0$. A reduction of this composition (the first projection of a pair) is given by the rule $\langle a, b\rangle; P_0 \Rightarrow a$.

Table 1 defines the basic set of rewritings underlying the Charity system. Rule (1) is the rewrite rule for "!" (the terminal map). This operation eliminates all the reductions that has performed in the composition sequence to the left of !. Rewrite rules (2) and (3) are the first and second projections on a pair, respectively. The projections on a pair a analogous to taking the first (P_0) or second (P_1) co–ordinate of a cartesian product. Finally, Rules (4) and (5) are the coproducts while (6) is the distribution rule for products.

(1)	$z; !$	$\Rightarrow$	$!$
(2)	$\langle x, y\rangle; P_0$	$\Rightarrow$	x
(3)	$\langle x, y\rangle; P_1$	$\Rightarrow$	y
(4)	$\langle z; b_0, x\rangle; \langle f\|g\rangle$	$\Rightarrow$	$\langle z, x\rangle; f$
(5)	$\langle z; b_1, x\rangle; \langle f\|g\rangle$	$\Rightarrow$	$\langle z, x\rangle; g$
(6)	$z; \langle x, y\rangle$	$\Rightarrow$	$\langle z; x, z; y\rangle$

Table 1: Rewrite rules for the basic combinators

In addition, operations on the datatypes are handled by a separate set of rewrite rules. The datatypes in the Charity system are divided into two classes, finite and infinite. Finite datatypes refer to structures that guarantee an end to the structure of the data when operations are performed on them. On the other hand, the data of an infinite datatypes may never end. Infinite datatypes are similar to the *streams* of standard functional languages. For example, one can define the natural numbers as a stream of numbers:

$0, 1, 2, 3, 4, \ldots$

For the finite datatypes there are three standard operations that manipulate the datatype (case^L (7), fold^L (8) and map^L (9)) are defined by the rewrite rules in

Table 2. The case operator views the head element of a datatype structure to determine an action. The fold operator can be viewed as head recursion call that rewrites the datatype structure to another datatype structure. Lastly, the map operator performs some action on each node of a the structure.

$$\begin{array}{r|rcl|}
\hline
(7) & \langle v_0; c_i, v_1\rangle; \text{case}^L\{f_1, \ldots, f_n\} & \Rightarrow & \langle v_0, v_1\rangle; f_i \\
(8) & \langle v_0; c_i, v_1\rangle; \text{fold}^L\{g_1, \ldots, g_n\} & \Rightarrow & \langle v_0; v_1\rangle; \langle \text{map}^{E_i}\{P_0, \text{fold}^L\{g_1, \ldots, g_n\}\}, \\
 & & & P_1\rangle; g_i \\
(9) & \langle v_0; c_i, v_1\rangle; \text{map}^L\{h_1, \ldots, h_m\} & \Rightarrow & \langle v_0; v_1\rangle; \text{map}^{E_i}\{h_1, \ldots, h_m, \\
 & & & \text{map}^L\{h_1, \ldots, h_m\}\}; c_i \\
\hline
\end{array}$$

Table 2: Rewrite rules for the finite datatypes

The rules for the infinite datatypes record^R (10), unfold^R (11) and map^R (12) are defined by Table 3. These rules allow operations on infinite datatypes similar to those of the finite datatypes.

$$\begin{array}{r|rcl|}
\hline
(10) & \text{record}^R\{f_1, \ldots, f_n\}; d_i & \Rightarrow & g_i \\
(11) & \text{unfold}^R\{g_1, \ldots, g_n\}; d_i & \Rightarrow & \langle f_i, P_1\rangle; \text{map}^{E_i}\{P_0, \text{unfold}^R\{f_1, \ldots, f_n\}\} \\
(12) & \text{map}^R\{h_1, \ldots, h_m\}; d_i & \Rightarrow & \text{map}^{E_i}\{h_1, \ldots, h_m, \text{map}^R\{h_1, \ldots, h_m\}\}; d_i \\
\hline
\end{array}$$

Table 3: Rewrite rules for the infinite datatypes

The remaining rewrite rules for the finite products are given in Table 4. Each map^{E_i} in the rewrite rules for the datatypes are substituted with the appropriate map in Table 4. The association between the two will be discussed fully in the **Compiling Categorical Combinators** section.

In a rewriting system, the order in which rewrite rules are applied in a reduction does not affect the answer. At each step in the reduction of a combinator expression

(13)	$\langle v_0, v_1\rangle; \text{map}^1$	$\Rightarrow$	v_0
(14)	$\langle v_0, v_1\rangle; \text{map}^{id}\{f\}$	$\Rightarrow$	$\langle v_0, v_1\rangle; f$
(15)	$\langle\langle v_0, v_1\rangle, v_2\rangle; \text{map}^{\times}\{f, g\}$	$\Rightarrow$	$\langle\langle v_0, v_2\rangle, \langle v_1, v_2\rangle\rangle; f \times g$

Table 4: Rewrite rules for the finite products

selecting one rewrite action is equivalent to another rewrite action, so long as rewrite rules are obeyed. For example, the reduction

$$\begin{aligned}
&w; \langle x, \langle y, z\rangle\rangle; P_1; P_0 && (6)\\
&\qquad \Rightarrow \langle w; x, w; \langle y, z\rangle\rangle; P_1; P_0 && (6)\\
&\qquad \Rightarrow \langle w; x, \langle w; y, w; z\rangle\rangle; P_1; P_0 && (3)\\
&\qquad \Rightarrow \langle w; y, w; z\rangle; P_0 && (2)\\
&\qquad \Rightarrow w; y
\end{aligned}$$

where the numbers to the right of each expression refer to a rewrite rule in the Charity system can be reduced equivalently by

$$\begin{aligned}
&w; \langle x, \langle y, z\rangle\rangle; P_1; P_0 && (3)\\
&\qquad \Rightarrow w; \langle y, z\rangle; P_0 && (2)\\
&\qquad \Rightarrow w; y
\end{aligned}$$

Clearly the first reduction requires fewer steps than the second. However, the point is that both reductions obtain the same result no matter what reduction route the system takes. A rewriting system simply specifies what is to be done, not how. The abstract machine presented in the next section uses rewrite rules to define the order that Charity expressions are reduced.

3 The Eager Abstract Machine

The Categorical Abstract Machine (CAM)[CCM85] is “a simple machine where categorical terms can be considered as code acting on a graph of values”. Categorical combinators have their foundation in category theory where composition, associativity and identity are the basic tools. Composition is denoted with the semicolon (;) and identity by (id). The addition of products (labelled $\langle x, y \rangle$) and projections (P_0, P_1) yields a Cartesian category. The idea behind the CAM machine is that categorical combinators can be interpreted as the instruction set in the abstract machine. The basic set of state transitions for the Charity system are derived from the CAM machine with the addition of transitions for the datatypes.

Evaluating combinators held in a graph is known as graph reduction and involves three basic steps:[Pey]

1. Unwind phase – finding the next reduction.
2. Reduction phase – Reducing a portion of a graph with an equivalent portion.
3. Update phase – Updating the node with the reduced portion.

The primary components of graph reduction are sharing and updating. A term is said to be shared if it is referenced more than once on the right hand side of an expression. As an example, consider the expression

$f(x) = x + x$, where x is some complicated expression.

In the above expression, x is considered to be shared since its value will be reference more than once. To enable sharing and avoid re–evaluation of a term, it is necessary to overwrite the original term with the evaluated value. The argument x will be evaluated once and further references to this argument will return the computed value of x.

In addition, terms can be evaluated in a “lazy” fashion where placeholders (called closures) are constructed to suspend evaluation of terms. This feature is useful as it reduces the unnecessary work performed by the evaluator. Consider the example below:

$g(x, y) = x$, where y is some complex expression.

The purpose of function g is to return the first argument, x. In a simple evaluator, both arguments are evaluated before the right hand side of the expression is executed. As a result, the evaluator performs unnecessary work by evaluating the argument y which is never used. Lazy evaluation attempts to avoid these sorts of inefficiencies by suspending evaluation until necessary.

Fundamental to the abstract machine are the state transitions that define how combinators are reduced. When reducing a combinator expression, rewrite rules dictates how to reduce the expression but they do not specify the sequence of the reduction. Abstract machines are more implementation oriented since the state transitions define the precise order reductions that need to be performed. This section gives a overview of the state transitions found in [CF92].

3.1 State transitions of the Charity system

The eager abstract machine is a three stack machine consisting of:

- **V –** The value stack to hold computations.
- **C –** The code stack containing the machine code.
- **D –** The dump stack to retain partial computations

The eager abstract machine performs reductions on a set of combinators through head rewriting, that is examining the top of each stack with pattern matching to determine the appropriate action to select. The state of the abstract machine is defined by the values contained in each of the three stacks (V, C, D) at any given time (between reductions). A reduction or state transition occurs when the state of the machine changes (ie: at least one of the stacks has been modified) and is designated by:

$$V \quad C \quad D \;\rightarrow\; V' \quad C' \quad D'$$

where the tuple (V, C, D) represents the "before" state and (V', C', D') the "after" state. For example, the transition for the first projection is

V	C	D		V'	C'	D'
$\langle v_1, v_0 \rangle$	$P_1.c$	d	$\rightarrow$	v_1	c	d

In the "before" state, the value stack V holds a pair and the head code stack has the instruction P_1. The above transition modifies the value stack and discards the instruction on the code stack to bring the machine to the "after" state. In this case, the dump remains unchanged and the machine carries on execution with the new "after" state. Evaluation is considered complete when the code stack and dump stack are empty and the result has been left on the value stack.

The state transitions are derived from the rewrite rules presented in the previous section. A few extra transitions are included to deal with administration details. Transition 1 is the terminal transition that resets the V stack. Transitions 2 and 3 are the projections of a pair while transitions 4 and 5 are the co–projections. A pair structure is built by transitions 6 to 8. Transition 9 handles the combinators such as constructors and infinite datatype operators where no action is performed.

Several of the transitions have been modified to better represent the state of the machine. Note in the transition tables how all items placed on the V stack are in reverse order. The reason for reversing the order is make the last item placed on the stack immediately accessible to the next transition.

	value	code	dump	$\to$	value	code	dump
1	v	$!.c$	d	$\to$	$!$	c	d
2	$\langle v_1, v_0\rangle$	$P_0.c$	d	$\to$	v_0	c	d
3	$\langle v_1, v_0\rangle$	$P_1.c$	d	$\to$	v_1	c	d
4	$\langle v_1, v_0.b_0\rangle$	$\langle c_0 \vert c_1\rangle.c$	d	$\to$	$\langle v_1, v_0\rangle$	c_0	$c(c).d$
5	$\langle v_1, v_0.b_1\rangle$	$\langle c_0 \vert c_1\rangle.c$	d	$\to$	$\langle v_1, v_0\rangle$	c_1	$c(c).d$
6	v	$\langle c_0, c_1\rangle.c$	d	$\to$	v	c_1	$\text{pr}_0(v, c_0, c).d$
7	v_1	$[\,]$	$\text{pr}_0(v, c_0, c).d$	$\to$	v	c_0	$\text{pr}_1(v_1, c).d$
8	v_0	$[\,]$	$\text{pr}_1(v_1, c).d$	$\to$	$\langle v_1, v_0\rangle$	c	d
9	v	$u.c$	d	$\to$	$u.v$	c	d
10	v	$[\,]$	$c(c).d$	$\to$	v	c	d
11	v	$[\,]$	$[\,]$	$\to$	HALT		

Table 5: State transition rules for the basic instructions

The datatype transitions associated with the rewrite rules are handled by transitions 12 – 14 for the finite datatypes and transitions 15 – 17 for the infinite datatypes. Finally map transitions are presented in Table 7 with transitions 18 – 20.

	value	code	dump	$\rightarrow$	value	code	dump
12	$\langle v_1, v_0.c_i \rangle$	$\text{case}^L\{f_1, \ldots, f_n\}.c$	d	$\rightarrow$	$\langle v_1, v_0 \rangle$	f_i	$c(c).d$
13	$\langle v_1, v_0.c_i \rangle$	$\text{fold}^L\{g_1, \ldots, g_n\}.c$	d	$\rightarrow$	$\langle v_1, v_0 \rangle$	$\text{map}^{E_i}\{P_0,$ $\text{fold}^L\{g_1, \ldots, g_n\}\}$	$\text{pr}_1(v1, g_i).c(c).d$
14	$\langle v_1, v_0.c_i \rangle$	$\text{map}^L\{h_1, \ldots, h_n\}.c$	d	$\rightarrow$	$\langle v_1, v_0 \rangle$	$\text{map}^{E_i}\{h_1, \ldots, h_n,$ $\text{map}^L\{h_1, \ldots, h_n\}\}.c_i$	$c(c).d$
15	$v.\text{unfold}^R\{f_1, \ldots, f_n\}$	$d_i.c$	d	$\rightarrow$	v	f_i	$\text{pr}_1(v, \text{map}^{F_i}\{P_0,$ $\text{unfold}^R\{f_1, \ldots, f_n\}\}).c(c).d$
16	$v.\text{case}^R\{g_1, \ldots, g_n\}$	$d_i.c$	d	$\rightarrow$	v	g_i	$c(c).d$
17	$v.\text{map}^R\{h_1, \ldots, h_n\}$	$d_i.c$	d	$\rightarrow$	v	$\text{map}^{E_i}\{h_1, \ldots, h_m,$ $\text{map}^R\{h_1, \ldots, h_n\}\}.d_i$	$c(c).d$

Table 6: State transition rules for the datatypes

	value	code	dump	$\rightarrow$	value	code	dump
18	$\langle v_1, v_0 \rangle$	map$^1.c$	d		v_0	c	d
19	v	map$^{id}\{f\}.c$	d	$\rightarrow$	v	f	$c(c).d$
20	$\langle\langle v_1, v_0 \rangle, v_2 \rangle$	map$^{\times}\{f, g\}.c$	d		$\langle v_2, v_1 \rangle$	g	pr$_0(\langle v_2, v_0 \rangle, f, c).d$

Table 7: State transition rules for finite datatypes

Recall the combinator expression

$w; \langle x, \langle y, z \rangle \rangle; P_1; P_0$

from Section 2. In a rewrite system, there may be more than one route in reducing a categorical combinator expression. In this section, the state transitions of the abstract machine specify exactly one route when reducing an expression to normal form. Table 8 illustrates a how the above categorical combinator expression is reduced by the eager abstract machine. The numbers to the right of each reduction correspond to the state transitions of the abstract machine found in Tables 5 to 7. Note that the answer is in reverse order. When the result is converted back into categorical combinators, the correct order is restored.

	V	C	D	
		$w; \langle x, \langle y, z \rangle \rangle; P_1; P_0$		(20)
$\Rightarrow$	w	$\langle x, \langle y, z \rangle \rangle; P_1; P_0$		(6)
$\Rightarrow$	w	$\langle y, z \rangle$	pr$_0(w, x, P_1; P_0)$	(6)
$\Rightarrow$	w	z	pr$_0(w, y, \_)$.pr$_0(w, x, P_1; P_0)$	(9)
$\Rightarrow$	$z.w$		pr$_0(w, y, \_)$.pr$_0(w, x, P_1; P_0)$	(7)
$\Rightarrow$	w	y	pr$_1(z.w, \_)$.pr$_0(w, x, P_1; P_0)$	(9)
$\Rightarrow$	$y.w$		pr$_1(z.w, \_)$.pr$_0(w, x, P_1; P_0)$	(8)
$\Rightarrow$	$\langle z.w, y.w \rangle$		pr$_0(w, x, P_1; P_0)$	(7)
$\Rightarrow$	w	x	pr$_1(\langle z.w, y.w \rangle, P_1; P_0)$	(9)
$\Rightarrow$	$x.w$		pr$_1(\langle z.w, y.w \rangle, P_1; P_0)$	(8)
$\Rightarrow$	$\langle \langle z.w, y.w \rangle, x.w \rangle$	$P_1; P_0$		(3)
$\Rightarrow$	$\langle z.w, y.w \rangle$	P_0		(2)
$\Rightarrow$	$y.w$			

Table 8: Reducing a Combinator Expression with an Abstract Machine

4 The Macro–coded Machine

Having specified the Charity system in the previous section, this paper now describes the implementation.

The implementation of the Charity system is divided into two levels, macro and micro. At the macro level, macro–instructions replace the state transitions of the eager abstract machine while micro–instructions perform stack operations. According the Hannan [Han91], abstract machines are "abstract" because pattern matching is used extensively in the construction and destruction of data. The construction and destruction of data refers to how data structures are build from atomic components and divided into atomic components, respectively. Hannan's approach is to use macro–instructions as identifiers that specify the action and micro–instructions to perform the action.

The benefit of this using Hannan's technique is the natural separation of the implementation into distinct and modular segments making system development a cleaner process. In addition, debugging is considerably easier with a two level approach.

4.1 The Macro–instructions set

Table 9 defines the basic macro–instructions, Table 10 and Table 11 the datatypes and Table 12 the administration instructions. The first column in the tables are the macro–instruction identifiers followed by an optional argument list. The second column is a brief description of the macro–instruction and column three gives a reference to the stat transition number from the tables (Table 1 to Table 4) in the last section. For example, in Table 9 the macro code for:

map_prod$\{f, g\}$	Map Product	(20)

means that the map_prod macro instruction takes two arguments (f and g) and

TERMINAL	The terminal combinator	(1)
ID	Identity combinator	
P0	First projection	(2)
P1	Second projection	(3)
PAIR	Evaluation of the first projection	(6)
PR0	Evaluation of the second projection	(7)
PR1	Recreate the pair structure	(8)
CONS	Constructors	(9)
CONT	Resume execution	(10)

Table 9: Macro instructions for the basic combinators

CASE $\{f_1, \ldots, f_n\}$	Function selection	(12)
FOLD $\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$	Fold over datatype	(13)
MAP_L $\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$	Map over datatype	(14)

Table 10: Macro instructions for the finite datatypes

is based on state transition rule (20) in Table 4. Much of the detail about how the macro–code operates has been omitted. At this level, the focus is on defining all the macro–instructions that make up the abstract machine. The micro–code described in the next section will define the specific actions each macro–instruction performs.

UNFOLD $\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$	Unfold over infinite datatype	(15)
RECORD $\{f_1, \ldots, f_n\}$	Infinite datatype record	(16)
MAP_R $\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$	Infinite datatype map	(17)

Table 11: Macro instructions for the infinite datatypes

MAP_1	Map value	(18)
MAP_ID $\{f\}$	Map identity	(19)
MAP_PROD $\{f,g\}$	Map product	(20)

Table 12: Macro instructions for the finite products

JUMP c, a	Resume execution at c with arguments a
PARM p	Setup a pointer to the parameter list
DEST d	A destructor
RET	Return from an evaluation

Table 13: Macro instructions for administration

Macro–instructions are executed by interpreting the macro–instruction identifier (eg: fold) and associating it with the appropriate micro–code. Macro–instructions are compiled down to a "linear" piece of code through the compiling process described shortly. Figure 3 provides a sample of how memory is organized after a combinator FOLD is compiled to macro–code. Note that the code store that represents other expressions are typically organized as a pointer to another structure/code. The parameters to FOLD ($\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$, the arguments and map substitutions) are stored in the memory address following each the instruction. A RET is stored at the end of a sequence of macro–instructions signals the end of execution for a particular piece of code.

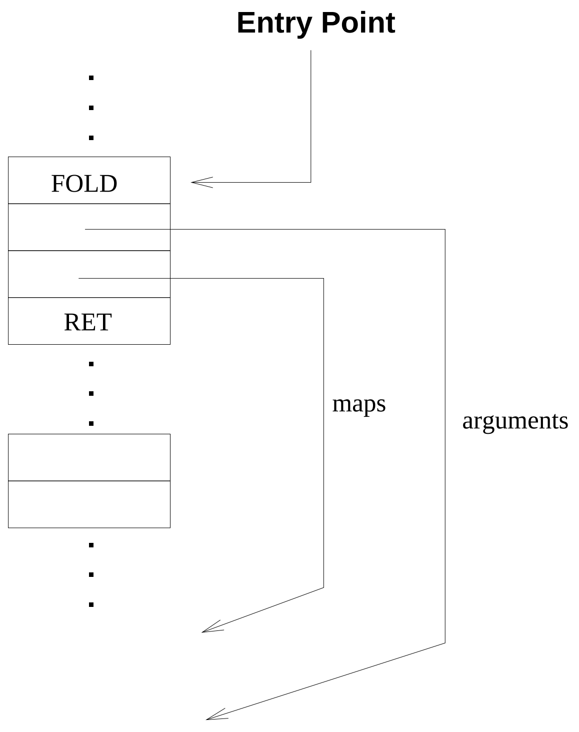

Figure 3: Sample code layout

Categorical combinator expressions are simply compiled to a linear sequence of macro–instructions. Consider the combinator expression:

$w; \langle x, \langle y, z \rangle \rangle; P_1; P_0$

when compiled, this combinator expression is stored in memory as shown in Figure 4.

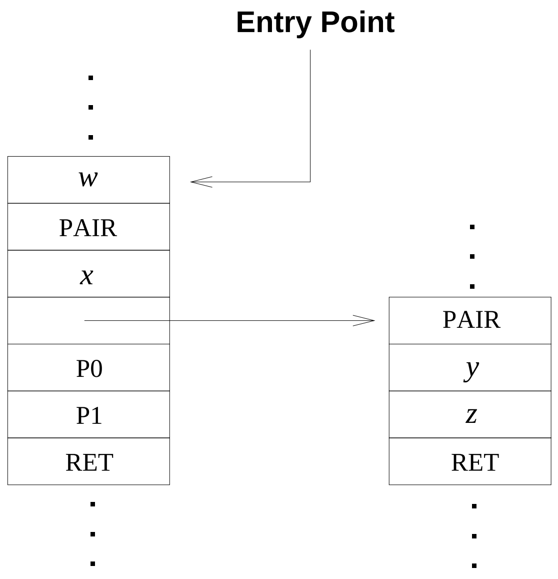

Figure 4: Macro–code representation of $w; \langle x, \langle y, z \rangle \rangle; P_1; P_0$

A sequence of macro–instructions defines a function, where calls to other functions are encoded into the code stream.

Categorical combinators are compiled down to one of the macro–instructions described above. Following some of the macro–instructions are pointers to the parameters that the machine code combination takes. Pattern matching has been limited to a tag that is needed to identify the instructions. The following words of memory are understood to be arguments of the instruction.

4.2 Compiling categorical combinators to Macro–instructions

4.2.1 The Compilation schemes

Compiling categorical combinator expressions is done through a set of compilation schemes. At the top level, a categorical combinator is either a combinator, a composition of combinators or an argument (abstraction) to a combinator expression. Table 14 defines how each categorical combinator is compiled. The concatenation of a string of combinators is represented by the "+" in the compilation scheme and the RET machine code identifier designates an end of a sequence of combinators. The composition of two combinators generates macro code for each combinator and concatenates the two together to build a linear sequence of machine code instructions. Each combinator can further be compiled down to a sequence of instructions described in the compiling schemes (Table 15). Lastly, categorical combinators can be an abstraction that must be substituted for the actual code at run time. The PARM machine code instruction and the offset into a table of abstract code that makes up the machine code for this case.

A combinator is divided into five classes:

- Distributive
- Inactive
- Active

```
compileSC($c$)is the machine code for the categorical combinator function $c$.

compileSC($c$)
    case ($c$) of

    $f; g$                          $\Rightarrow$  compileSC($f$) + compileSC($g$)
                                                   where $f$ and $g$ are categorical combinators.
  | $c\,\{f_1, \ldots, f_n\}$       $\Rightarrow$  compileC($c\,\{f_1, \ldots, f_n\}$
                                                   where $c$ is the combinator identifier and $f_1, \ldots, f_n$
                                                   the arguments for the combinator.
  | $n$                             $\Rightarrow$  PARM + $n$ + RET
                                                   where $n$ is the positional value of the abstraction.
```

Table 14: The compileSC compilation scheme

- Ractive
- Cdef

A combinator is distributive if it is a terminal (!), identity (id) or a pair and handled by the compilation rules in Table 16. Inactive combinators refer to constructors and right datatype (infinite) operators that do not act on the value graph in the machine. In Table 17 constructors are compiled with the identifier CONS and the constructors positional value. Active combinators such as the projections maps (P_0, P_1), map combinators (map^1, map^{id}, map$^\times$), and destructors do some manipulation on the value graph at run time (see Table 18). Cdefs are functions expressed as categorical combinator expressions that compile down to a JUMP plus the jump location and its arguments.

```
compileC [$c\ x_1, \ldots, x_n$] is the compilation of a combinator.

compileC($c\ \{x_1, \ldots, x_n\}$)   $=$
    case ($c$) of

    Distributive      $\Rightarrow$  compileDist ($c\ \{x_1, \ldots, x_n\}$)
                                     where $c \in \{!, \mathrm{id}, \mathrm{pair}\}$ and $x_1, \ldots, x_n$
                                     are the arguments for pair.
  | Inactive          $\Rightarrow$  compileInactive ($c\ \{x_1, \ldots, x_n\}$)
                                     where $c$ is a constructor or a right datatype operator and
                                     $x_1, \ldots, x_n$ are the arguments for the right datatype.
  | Active            $\Rightarrow$  compileActive ($c\ \{x_1, \ldots, x_n\}$)
                                     where $c \in \{\mathrm{map\_1}, \mathrm{map\_id}, \mathrm{map\_prod},$
                                     $p_0, p_1, \mathrm{id}, \}$ and $x_1, \ldots, x_n$ are the arguments.
  | Cdef              $\Rightarrow$  JUMP + $c$ + $\{x_1, \ldots, x_n\}$
                                     where $c$ is the location and $x_1, \ldots, x_n$ the arguments.
```

Table 15: The compileC compilation scheme

```
compileDist($c$)   $=$
    case ($c$) of

    id            $\Rightarrow$  RET
  | PAIR($f$, $g$)  $\Rightarrow$  PAIR + compileSC($f$) + compileSC($g$) + RET
  | !             $\Rightarrow$  TERMINAL
```

Table 16: Compiling Distributive combinators

```
compileInactive($c\ \{x_1, \ldots, x_n\}$)

    case ($c$) of
    constructor($n$)     $\Rightarrow$  CONS + $n$ + RET
                                   where $n$ is the constructor position.
  | right_op($c$)        $\Rightarrow$  RIGHT_OP + c
                                   where $c$ is the code.
```

Table 17: Compiling Inactive combinators

```
compileActive($c$)                          $=$
    case ($c$) of

    $P_0$                                   $\Rightarrow$  p0
|   $P_1$                                   $\Rightarrow$  p1
|   map$^{id}$                              $\Rightarrow$  map_id
|   map$^1$                                 $\Rightarrow$  map_1
|   map$^\times$                            $\Rightarrow$  map_prod
|   Other($\{s_1, \ldots, s_n, f_1, \ldots, f_n\}$)   $\Rightarrow$  JUMP + $\{s_1, \ldots, s_n\}$ + $\{f_1, \ldots, f_n\}$ + RET
```

Table 18: Compiling Active combinators

4.2.2 Compiling "Maps"

Recall the state transitions for the datatype operators FOLD, UNFOLD, MAP_L, and MAP_R where a map^{E_i} appeared on the right hand side of the transition. These maps are the basic components in evaluating datatype operators.

$$\begin{array}{lllll}
\text{data} & L(A_1, \ldots, A_m) & \rightarrow & S & = \\
 & c_1 \colon E_1(A_1, \ldots, A_m, S) & \rightarrow & S & \\
 & \vdots & & & \\
 & c_n \colon E_1(A_1, \ldots, A_m, S) & \rightarrow & S &
\end{array}$$

Given a constructor and the FOLD map, this can be re–written as above, where map^{E_i} is determined by the type E_i in the datatype $L(A_1, \ldots, A_m)$. In the eager abstract machine, the replacement for the map^{E_i} with the appropriate combinator is performed at run–time because the constructor c_i determines the map^{E_i} selection. The eager machine queries the symbol table for the proper "map substitutions" and generates the code accordingly. Such an expensive replacement operation can be eliminated from the run time system by compiling the map^{E_i} code before the abstract machine executes the program.

One solution is to compile each constructor template with substitutions already resolved. For instance, given the combinator FOLD$_{nat}\{f_1, \ldots, f_n\}$, the map^{E_i} for each constructor in the fold (ie: zero and succ) are compiled down to the form shown in Figure 5.

The FOLD is the machine code identifier (tag) and the following 2 words of memory are pointers to the arguments ($f_1, \ldots, f_n$) and map^{E_i} ($s_1, \ldots, s_n$) tables, respectively.

At run time, a constructor record contains a offset into the tables where the code (either a parameter of a map^{E_i}) is loaded by calculating the address of the code.

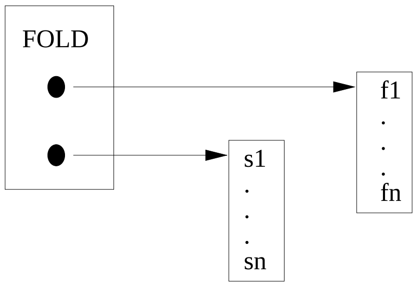

Figure 5: Machine layout of a sample compile

For example, given the constructor succ (has positional value of 1) of the natural numbers datatype and the fold$\{f_1, \ldots, f_n\}$ combinator, the machine would:

1. load address of map^{E_i}
2. add the offset of the constructor position (ie: 1)
3. jump to the code that the entry in the table points to

An Example of map substitutions

The definition of a binary tree is defined in Charity term logic by :

```
data btree(A,B) ->  T =
     leaf: A             ->  T
  |  node: (T * (B * T)) ->  T.
```

that says the datatype btree has 2 constructors (leaf and node), where the leaf takes a parameter A and the node has a left and right subtree (T) with a parameter B. This means that the datatype *btree* contains the objects:

$$\begin{aligned} E_{leaf}(A,B,T) &= A \\ E_{node}(A,B,T) &= (T \times (B \times T)) \end{aligned}$$

The corresponding map^{E_i}'s are

$$\begin{aligned}\text{map}^{E_{leaf}}(f,g,h) &= \text{map\_id}\ \{f\}\\ \text{map}^{E_{node}}(f,g,h) &= \text{map\_prod}\ \{h, \text{map\_prod}\{f,h\}\}\end{aligned}$$

Given a machine state, one rewrites to another state with parameters substituted.

$$\begin{aligned}\langle v_1, v_0; \text{leaf}\rangle; \text{fold\_btree}\{f,g\} &\Rightarrow \langle p_1, \text{map}^{E_{leaf}}\{p_0, p_0, \text{fold\_btree}\{f,g\}\}\rangle; f\\ &\Rightarrow \langle p_1, \text{map}^{id}\{p_0\}\rangle; f\end{aligned}$$

The above rule is interpreted as having a constructor (leaf) and a fold_btree combinator. The combination of these two items can be rewritten as:

$\langle \text{map}^{E_{leaf}}\{p_0, p_0, \text{fold\_btree}\{f,g\}\}, p_1\rangle; f$

However map^{E_i} can also be thought of as a function that takes three parameters (p$_0$ if the object is non–recursive and fold_btree if the object is recursive).

The substitutions for map^{E_i} are known at compile time, however, what is not know is the particular substitution to select. The map^{E_i} is determined by the constructor seen on the first projection of a pair. To improve the performance of the machine, the correspond map^{E_i}'s for each constructor are compiled into a list of map^{E_i}'s which is selected by adding the constructor offset to the beginning of a table. This is an improvement over the existing method of creating the map^{E_i} at runtime.

5 The Micro–coded Machine

Once categorical combinator expressions are compiled down to macro–code, they are ready for execution. As mentioned, each macro–instruction is composed of micro–instructions that manipulate the machine state. The machine described in this section interprets each macro–instruction and executes a series of micro–instructions. At an even lower level, micro–instructions are composed of

- assignment statments
- indirections
- simple arithmetic (addition and subtraction)
- flow control

Two important properties of the Charity language to notice when evaluating evaluating charity functions at the machine level are:

1. the language has strong type checking
2. programs are guaranteed to terminate.

At the machine level, a strong type checking system means that all micro–instructions will perform a legal action. Type checking guarantees that a micro–instruction can expect that required values are available when the instruction is executed. As a result, many of the error checks can be omitted from the runtime system. Furthermore, all Charity programs will terminate after a fixed amount of processing including programs that use infinite datatypes. Recursion in Charity is done by the FOLD and MAP^L operators. For the infinite datatypes, programs terminate because Charity processes the datatypes one level at a time by the UNFOLD and MAP^R operators. By guaranteeing termination, the machine will never enter into an infinite loop and an answer will always be returned.

5.1 Micro–coded architecture

The state of the machine is defined by the tuple:

$\langle PC, V, D, H, A, R0, R1\rangle$

where

- PC is the program counter
- V is the value pointer
- D is the dump stack
- A is the abstraction stack
- H is the heap
- $R0$ a general purpose register
- $R1$ a general purpose register

A transition in the micro–coded CHARM machine is defined by:

$$\langle PC, V, H, D, A, R0, R1\rangle \quad \rightarrow \quad \langle PC', V', H', D', A', R0', R1'\rangle$$

PC – The program counter

The register PC points to the head of the code stream the machine is currently executing. Valid items on the code stream are either macro–instruction labels (eg: FOLD) or an indirection to some other code store. A JUMP or changing the control flow of a program is done by assigning a new value to PC (usually done by incrementing the PC register for the next instruction).

V – The value register

The V register points to items in a heap structure that has been created from transitions in the machine.

D – The dump stack

The purpose of the dump stack is to allow partial computations to be stored until the machine has finished computing a earlier result. The register D references items on the top of the dump stack.

A – The abstraction stack

Abstractions are stacked onto the abstraction stack. Functions that require abstractions simply offset into the abstraction stack to retrieve the address the machine "jumps" to. Once a function has finish processing, the values in the abstraction stack are poped off.

$R0$ and $R1$ – General purpose registers

The general purpose registers $R0$ and $R1$ are used in intermediate calculations and parameter passing between macro–instructions.

5.2 The Heap

The heap is a contiguous block of memory that holds the computed values of the machine. Items in the Heap consist of:

1. The Heap identifier. Usually a PAIR, CONS (constructor) or a right datatype operator (eg: UNFOLD, MAP^R, or RECORD).

2. Arguments to the above heap item. For example, in the heap item PAIR, 2 words of memory, each pointing to a projection of the pair are allocated in the heap. For constructors, 3 extra words of memory for the constructor position, name and an indirection to next element in heap. Figure 6 shows an example of how the pair is represented in the heap of the abstract machine.

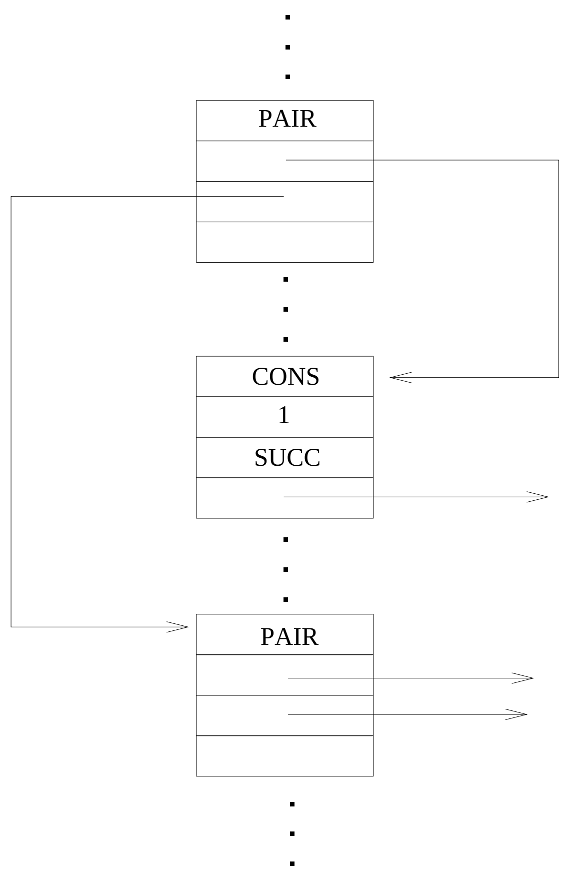

Figure 6: Organization of the Heap

The heap is a fixed structure that does not retain its values between evaluations. That is, each call to the abstract machine to execute a combinator expression by the CHIRP user interface will reset the heap when evaluation begins. Here garbage collection is not needed since at the beginning of each evaluation, a "new" heap is used.

5.3 Sharing Heap Items

Frequently, structures in the heap are shared. Recall that a structure is shared when there are multiple references to it. In the eager machine the sharing of structures is common. To enable sharing without creating side effects, the parent of the shared item must be copied to another area in the heap. This newly created structure can then be modified to reflect the necessary changes with no loss of sharing. The implications of sharing means that in certain situations heap items can be overwritten with new values to save heap storage. In other cases, the heap item must be duplicated and changes must be made to the newly built structure. For sharing to work properly in the eager machine, two rules must be followed in dealing with sharing:

1. Heap objects referenced by the value register are not reusable. That is, the machine is not permitted to overwrite heap objects that are referenced by the value register.
2. Heap objects referenced in the dump stack can be immediately reused by the machine.

If the machine needs to change the structure of a heap item that is referenced by the V register, it must do so by copying the item and making the changes to the new item. Items referenced by the V register are considered to be shared. Figure 7 shows how a heap item is copied without loss of sharing on the existing structure. The dashed line is the new heap item that the V register points to after copying.

On the other hand, heap items referenced by the dump stack D can be overwritten with new values. An example of reusing heap space is shown by the state transition rules for building a PAIR. Storage is created in the heap when the PAIR must evaluate its first component and save the second for future computation. When both components of a pair have been computed the heap space that was used in to save the partial computation can be overwritten with the PAIR structure.

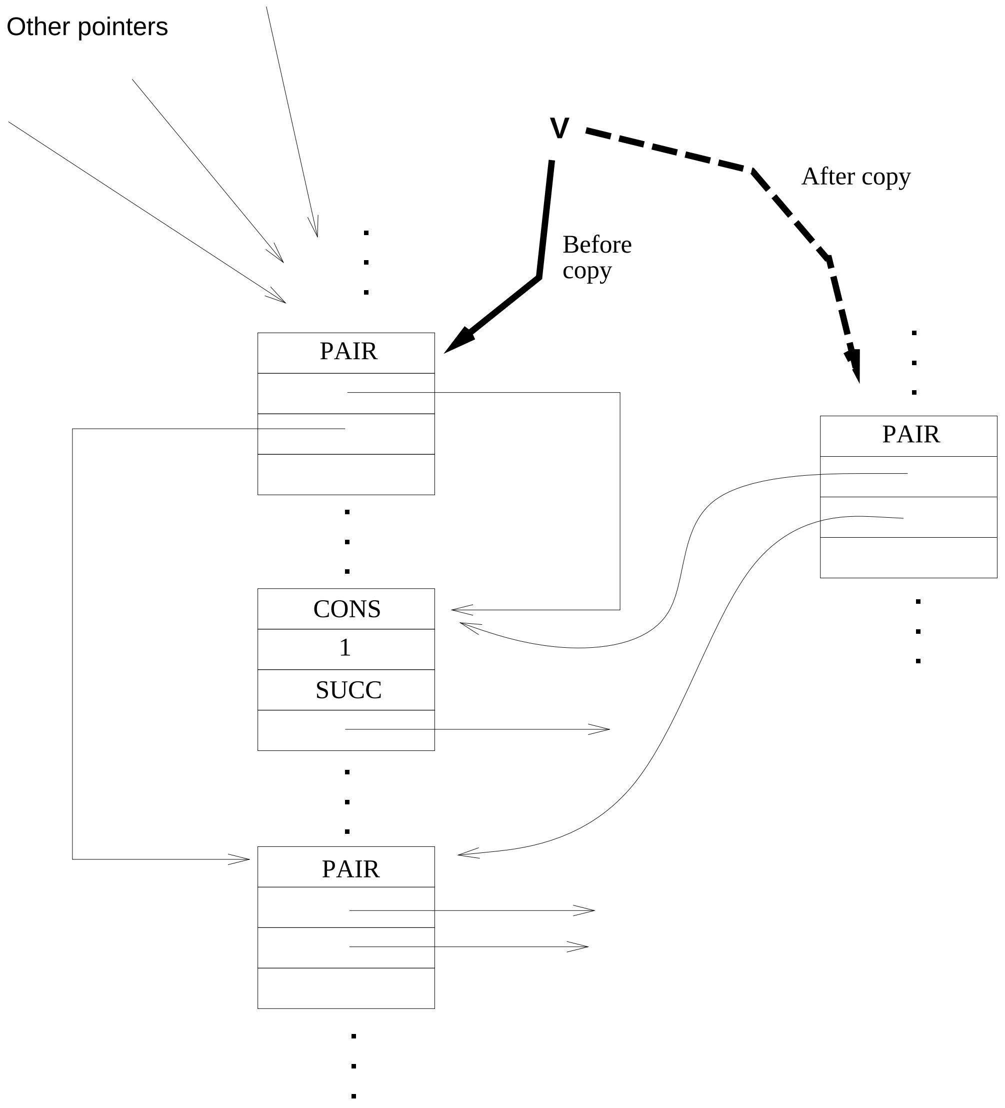

Figure 7: Copying a Shared Heap Item

5.4 The Micro–coding of Macro–instructions

The micro–instructions specify the action associated with each macro–instruction. The following notation will be used in defining the micro–instructions:

- (R) – means get the contents of R.
- R – means an indirection to the contents of R

In addition, several support functions are provided:

- alloc(n, R) – allocate n items of heap space, returning a pointer in R
- push(R, D) – Push the contents of register R to the dump D
- pop(D, R) – Pop an item from D dump to register R
- pushA(A, n) – load the abstractions from table A onto abstraction stack, leaving the count in n.

The following is a listing of the micro–instructions that make up each macro–instruction. The macro–instruction name is presented in bold with a short description of the actions. Following the description is the state transition rule (where applicable) of the macro–code. Finally, the micro–instructions for each macro–instruction is listed.

TERMINAL

The terminal instruction resets the V register.

$$(1) \quad (v, !.c, d) \quad \rightarrow \quad (!, c, d)$$

```
V = TERMINAL
```

P0

P_0 selects the first co–ordinate of a pair (The first projection). Note that the values in a pair are reversed.

$$(2) \quad (\langle v_1, v_0 \rangle, P_0.c, d) \quad \rightarrow \quad (v_0, c, d)$$

```
R0 = (V + 2)       ; Get the pointer to first projection
V  = R0            ; Save the value to the V register
```

P1

P_1 selects the second co–ordinate of a pair (The second projection).

$$(3) \quad (\langle v_1, v_0 \rangle, P_1.c, d) \quad \rightarrow \quad (v_1, c, d)$$

```
R0 =  V + 1        ; Get the pointer to first projection
V  = R0            ; Save the value to the V register
```

PAIR$\{c_0, c_1\}$

Construct a pair structure in the heap. First evaluate the second project, storing the first on the heap.

$$(6) \quad (v, \langle c_0, c_1 \rangle.c, d) \quad \rightarrow \quad (v, c_1, \mathrm{pr}_0(v, c_0, c).d)$$

```
alloc(3, R0)        ; get 3 words of heap to store pair
R0     = v          ; <v1, v0>
R0+1   = PC + 1     ; c0
R0+2   = (PC) + 3   ; c -- save PC for continuation
push(R0, D)         ; store pr0(<v1, v0>, c0, c)

PC = (PC) + 2       ; Jump to c1
```

PR0$\{v, c_0, c\}$

Continue evaluation of the first projection of a pair. Store the computed value in the heap.

$$(7) \quad (v_1, [\,], \mathrm{pr}_0(v, c_0, c).d) \quad \to \quad (v, c_0, \mathrm{pr}_1(v_1, c).d)$$

```
R0   = PC + 1       ; pr0(v, c0, c)
R1   = v            ; swap v1 with c0
V    = R0           ; load v
PC   = R0 + 1       ; load c0 into PC
R0+1 = R1           ; save v1
push(R0, D)         ; create pr1(v1, c) in heap
push(PR1, D)
```

PR1$\{v1, c\}$

Create the fully evaluated pair structure as item in the heap. Reuse the heap storage space that was used to hold the partial computations.

$$(8) \quad (v_0, [\,], \mathrm{pr}_1(v_1, c).d) \quad \to \quad (\langle v_1, v_0\rangle, c, d)$$

```
R0   = PC + 1      ; load pr1(v1, c)
R0   = PAIR        ; create <v1, v0> reuse heap storage
PC   = R0 + 2      ; Set PC to resume at c
R0+2 = V           ; <v1, v0>
V    = R0          ; set V to hold <v1, v0>
```

CONS$\{p, n\}$

Instantiate a constructor in the heap. A construction is composed of an identifier, positional value, pointer to the constructor name and an indirection to the next heap item.

$$(9) \quad (v, u.c, d) \quad \rightarrow \quad (u.v, c, d)$$

```
alloc(4, R0)       ; get heap space for constructor
R0   = CONS        ; assign the tag CONS
R0+1 = PC + 1      ; the positional value
R0+2 = PC + 2      ; the constructor name
R0+3 = V           ; the next value is in V
V    = R0          ; set the new V value
```

CONT$\{c\}$

Resume evaluation code that has been suspended.

$$(10) \quad (v, [\,], c(c).d) \quad \rightarrow \quad (v, c, d)$$

```
PC = PC + 1          ; resume execution at c
```

CASE$\{f_1, \ldots, f_n\}$

Examine the argument and build a new structure with the construction stripped away. Load the address of the function f_i into the program counter and begin execution.

$$(12) \quad (\langle v_1, v_0.c_i\rangle, \mathrm{case}^L\{f_1, \ldots, f_n\}.c, d) \quad \rightarrow \quad (\langle v_1, v_0\rangle, f_i, c(c).d)$$

```
R1   = V + 2         ; Examine the constructor
alloc(3, R0)         ; Copy the pair stripping constructor
R0   = PAIR          ; Heap identifier
R0+1 = V + 1         ; <v1, _>
R0+2 = R1 + 4        ; strip cons --> <v1, v0>
V    = R0            ; V = <v1, v0>
push(PC+2, D)        ; save return address
push(CONT, D)        ;  as a continuation

R0   = PC + 1        ; load the argument list {f1,...,fn}
R1   = R1 + 1        ; load the positional value of constructor
PC   = R0 + R1       ; calculate the appropriate offset
```

FOLD$\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$

Load the appropriate map^{E_i} into the machine for execution.

$$(13) \quad (\langle v_1, v_0.c_i\rangle, \mathrm{fold}^L\{g_1, \ldots, g_n\}.c, d)$$

$$\rightarrow \quad (\langle v_1, v_0\rangle, \mathrm{map}^{E_i}\{ \begin{matrix} P_0, \\ \mathrm{fold}^L\{g_1, \ldots, g_n\} \end{matrix} \}, \mathrm{pr}_1(v_1, g_i).c(c).d)$$

```
R1   = V + 2        ; Examine the constructor
R1   = R1 + 1       ; load constructor position
R0   = PC + 1       ; load argument list (g1,...,gn)
R0   = R0 + R1      ; Calculate the address of gi

alloc(3, R1)        ;
R1   = PAIR         ; create pr1(v1, c)
R1+1 = V + 1        ; store v1
R1+2 = R0           ; store gi
push(R1, D)         ; push pr1(v1, c) onto dump
push(PR1, D)

R1   = V + 2        ; Examine constructor again
alloc(3, R0)        ;
*R0  = PAIR         ; Copy the pair structure <v1, v0>
R0+1 = V + 1        ; v1
R0+2 = R1 + 4)      ; v0
V    = R0           ; Set v = <v1, v0> (constructor stripped)

R1   = R1 + 1       ; load constructor position
R0   = PC + 2       ; load Map_Ei list (s1,...sn)
PC   = R0 + R1      ; calculate address of Map_Ei
```

MAP_L$\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$

Load map^{E_i} into the program counter.

$$(14)\quad (\langle v_1, v_0.c_i\rangle, \text{map}^L\{h_1,\dots,h_n\}.c, d)$$
$$\rightarrow\quad (\langle v_1, v_0\rangle, \text{map}^{E_i}\{\begin{array}{l} h_1,\dots,h_n, \\ \text{map}^L\{h_1,\dots,h_n\}\end{array}\}.c_i, c(c).d)$$

```
R1   = V + 2       ; Examine the constructor
alloc(3, R0)       ; Copy the pair, stripping constructor
*R0  = PAIR        ; Heap identifier
R0+1 = V + 1       ; <v1, _>
R0+2 = R1 + 4      ; strip constructor --> <v1, v0>
V    = R0          ; v = <v1, v0>
alloc(4, R0)       ; Copy the constructor
R0   = R1+2        ; constructor name
R0+1 = R1+1        ; positional value of constructor
R0+2 = R1          ; CONS identifier
push(R0, D)        ; push the constructor onto the heap
push(CONT, D);

R0   = PC + 2      ; load the mapE_i list (s1,...,sn)
R1   = R1 + 1      ; load the positional value of constructor
PC   = R0 + R1     ; calculate the address of mapE_i
```

UNFOLD$\{f_1,\dots,f_n,s_1,\dots,s_n\}$

Similar to the FOLD case where the f_i is loaded and a pair continuation is pushed to the dump stack.

$$
\begin{array}{rl}
(15) & (v.\text{unfold}^R\{f_1,\ldots,f_n\}, d_i.c, d) \\
\to & (v, f_i, \text{pr}_1(v, \begin{array}{l}\text{map}^{F_i}\{P_0, \\ \text{unfold}^R\{f_1,\ldots,f_n\}\end{array} \}).c(c).d)
\end{array}
$$

```
                    ; Destructor passed in R0
R0   = R0 + 1       ; load destructor position
R1   = V + 1        ; load argument list (f1,...,fn)
PC   = R1 + R0      ; calculate address of fi
push(R1, D)         ; temporarily save fi
R1   = V + 2        ; load Map_Ei list (s1,...,sn)
R1   = R1 + R0      ; calculate address of si

alloc(3, R0)        ; create pr1(v, map_Ei)
R0   = V            ; v
R0+1 = R1           ; R1
pop(D, R1)          ; Restore fi
push(R0, D)         ; push pr1(v, map_Ei) onto dump
push(PR1, D)
V    = V + 3        ; strip UNFOLD label
```

RECORD$\{f_1,\ldots,f_n\}$

Destruct the infinite list at one level and execute the appropriate f_i.

$$(16) \quad (v.\text{RECORD}\{g_1,\ldots,g_n\}, d_i.c, d) \quad \to \quad (v, g_i, c(c).d)$$

```
                    ; Destructor passed in R0
R1   = PC + 1       ; load argument table (g1,...,gn)
R0   = R0 + 1       ; load destructor positional value
PC   = R1 + R0      ; calculate gi
```

MAP_R$\{f_1, \ldots, f_n, s_1, \ldots, s_n\}$

$$\begin{aligned} (17)\quad & (v.\text{map}^R\{h_1, \ldots, h_n\}, d_i.c, d) \\ \rightarrow\quad & (v, \text{map}^{E_i}\{h_1, \ldots, h_m, \text{map}^R\{h_1, \ldots, h_n\}\}.d_i, c(c).d) \end{aligned}$$

```
                   ; destructor passed in R
alloc(5, R1);      ;
R1   = R0 + 2      ; destructor name
R1+1 = R0 + 1      ; destructor position
R1+2 = R0 + 2      ; destructor tag
push(R1, D)        ; push the destructor on dump
push(CONS, D);

R1   = PC + 2      ; load Map_Ei table
R0   = R0 + 1      ; load destructor position
PC   = R1 + R0     ; calculate address of Map_Ei
```

MAP_1

The action for MAP_1 is the same as P0

$$(18)\quad (\langle v_1, v_0\rangle, P_0.c, d) \quad\rightarrow\quad (v_0, c, d)$$

```
R0 = V + 2          ; Get the pointer to first projection
V  = R0             ; Save the value to the V register
```

MAP_ID$\{f\}$

MAP_ID specifies executes the function f.

$$(19) \quad (v, \text{MAP}^{id}\{f\}.c, d) \quad \rightarrow \quad (v, f, d)$$

```
PC = (PC) + 1       ; Increment the program counter to point to f
```

MAP_PROD$\{f, g\}$

Distribute a value over a pair by creating new pair structures for f and g and loading the address of g into the program counter.

$$(20) \quad (\langle v_1, v_0\rangle, \text{MAP}^{\times}\{f, g\}.c, d) \quad \rightarrow \quad (\langle v_2, v_1\rangle, g, \text{pr}_0(\langle v_2, v_0\rangle, f, c).d)$$

```
R0     = V + 2     ; load <v1, v0>
R1     = R0 + 2    ; load v0
alloc(3, R0)       ; get 3 words of heap to store pair
R0     = PAIR      ; Store a pair initially <_,_>
R0+1   = v + 1     ; <v2, _>
R0+2   = R1        ; <v2, v0>

alloc(3, R1)       ; get 3 words to store a continuation
R1     = R0        ; push <v2, v1> onto heap
R1+1   = PC + 1    ; push f onto heap
R1+2   = PC + 3    ; save the return address
push(R1, D)        ; push pr0(<v2, v1>, f, c) onto dump
push(PR0, D)

R0     = V + 2     ; load <v1, v0>
R1     = R0 + 1    ; load v1
alloc(3, R0)       ; get 3 word of heap to store pair
R0     = PAIR      ; Store a pair initially <_,_>
R0+1   = V + 1     ; push <v2, _>
R0+2   = R1        ; <v2, v1>

V      = R0        ; Set V to point to <v2, v1>
PC     = PC + 2    ; Jump to g
```

DEST$\{d\}$

When a destructor is encountered in the code stream, the positional value of the destructor is passed to the right datatype operator in the register $R0$.

```
R0   = PC + 1       ; Pass the positional value in R0
push(PC+2, D)       ; Save return address
push(CONT,D)
PC   = V + 1        ; load the right data type operator
V    = V + 2        ; strip off the operator from V
```

JUMP$\{c, a\}$

Functions are executed by JUMPing to the address where their macro–instructions are stored. The abstraction table must be loaded into the abstraction stack and a marker placed on the dump stack to “pop” the values off.

```
loadA(PC+2, R0)     ; load the abstraction table into A stack
push(R0, D)         ; push marker to pop abstractions.
push(POPA, D)
PC     = PC + 1     ; load the function into PC
```

PARM$\{n\}$

Abstractions in functions are referenced by a PARM identifier in the code stream. The address of the abstraction is calculated by using an offset on the A register.

```
push((PC)+2, D)     ; push return address
push(CONT, D)
R0   = A            ; load address of abstraction stack
R1   = PC + 1       ; load offset
PC   = R0 + R1      ; calculate the abstraction address
```

POPA$\{n\}$

Pop n items off the abstraction stack

```
R0   = D           ; load address of D
R1   = PC + 1      ; load number of abstractions
D    = R0 + R1     ; pop the items
```

5.5 Abstractions

Abstractions to functions pose an interesting problem with scoping in the machine. That is, if a chain of functions exist where each requires an abstraction from the calling function, a method must be developed for abstractions that pass abstractions as parameters. For example, consider the chain of incomplete Charity functions defined below:

```
def foo {h, i} (...) =
        i(...).

def foo1 {f, g} (...) =
        foo{a => func3(), b => f(...)} (...).

def foo2 (...) =
        foo1{a => func1(a), b => func2(b)} (...).
```

The machine must keep track of all abstractions and the functions they are associated with. For example function `foo1` takes two abstractions (f and g). When `foo1` is called from `foo2` the parameters for f and g are the functions `func1` and `func2`, respectively. Both functions `func1` and `func2` are pushed onto the abstraction stack for access in `foo1`. When `foo1` calls `foo`, the parameters `func3` and the abstraction f are pushed onto the abstraction stack. Eventually, function `foo` will access the second abstraction i only to find that it is another abstraction. What the machine needs to do is copy the pointer to the abstraction in the previous level. For the above case, the pointer to function `func3` is pushed onto the abstraction stack and not the abstraction f. Figure 8 gives an illustration of how the abstraction stack looks like after the abstractions have been pushed onto the stack and a call has been issued to `foo`.

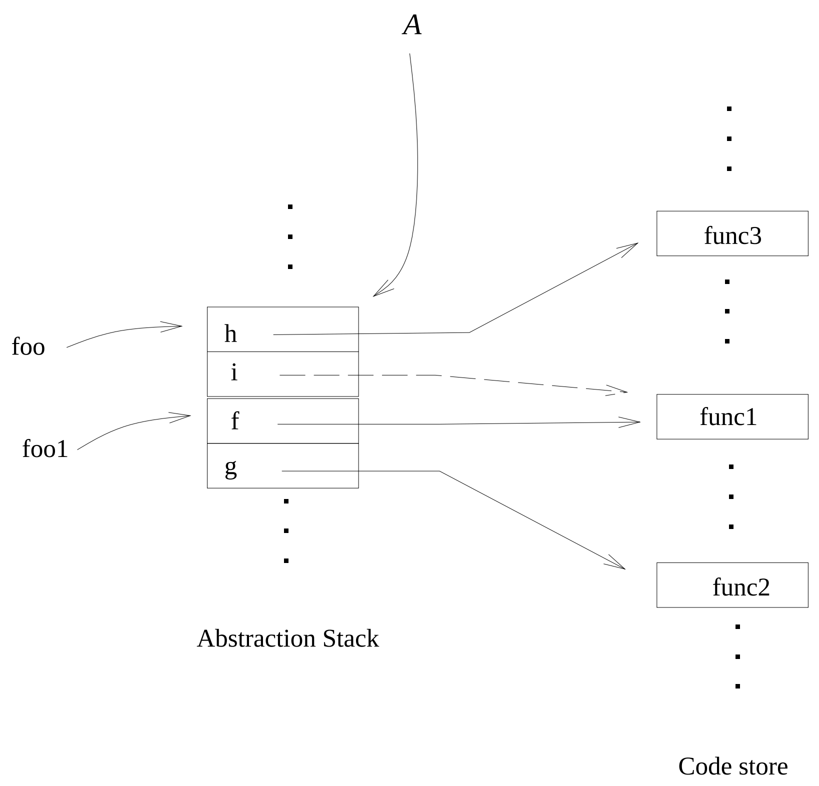

Figure 8: Sample representation of Abstractions

Note that the dashed line from (i) in Figure 8 means that a pointer from the previous level has been copied. Entering a function causes abstractions to be placed on the stack with abstractions parameters already resolved. At most, the machine need only look one level deep to resolve an abstraction, thus solving the scoping problem of abstractions.

On exit of the function, the items on the abstraction stack are poped off. When a function calls another function, a marker to placed on the dump stack so that when the called function is finishes execution, the abstraction stack is cleaned up.

6 Future work

Though this project implements a reasonable eager machine in C, there is still room for improvement. Hermann's CHARM machine (see Appendix A) uses lazy reduction in the evaluation of programs. In a full blown lazy system, updating, sharing, and suspensions all contribute to making the system more efficient. It appears in Hermann's implementation of a lazy machine that "laziness" is only useful with the infinite datatypes. With the finite datatypes, the machine spends too much time building suspensions only to load the suspension back into the machine for immediate evaluation.

Perhaps combining Hermann's CHARM machine with some sort of abstract interpretation that eliminates the building of suspensions at compile time will improve the performance of the system. In abstract interpretation, the compiler attempts to proved that arguments are are needed by the function and should be evaluated immediately.

Other areas for work remaining for the CHARM project are:

- Port the entire system to another computer platform
- Measure performance of the system in comparison to other functional languages
- Implement a lazy system for the infinite datatypes
- Implement garbage collection on the heap

Finally, the categorical combinator level should be eliminated so that Charity programs are compiled directly to "machine code". This will reduce the effort need to maintain two sets of symbol tables in the machine.

A Models of Abstract Machines

A.1 CHARM abstract machine

The original CHARM machine is written in SML by M. Hermann [Her92] uses graph reduction to reduce unnecessary work by the evaluator. A *closure* provides a method of representing values and code in a uniform manner. A closure is a structure that holds a pointer to the code and its arguments. Self updating closures contain an update marker with the address of the node to overwrite. When a closure is entered, the update information stored in the closure ensure that the result overwrite the value node and subsequent references to the closure simply returns the computed answer.

In [Her92] expressions were found to be shared in only three ways that can be detected at run–time. In brief they are:

1. Multiple references to a term on the right hand side of a rewrite rule.
2. Terms shared by repeated iterations of the FOLD and UNFOLD combinators.
3. Terms inherit sharing.

Hermann's abstract machine attempts to detect sharing and updating are required in a closure. A closure in this machine is represented by the tuple (v, c, u), where

- v is a pointer to the value in the heap
- c is a pointer to the code
- u is the update marker with the address to update

When sharing is detected and an update is required, the update information in the closure is annotated the marker and address of the value to update. If sharing is inherited, each child closure to the parent closure is annotated with the update information. Recall that in the eager machine arguments to a functions are evaluated to normal form. In the CHARM machine, lazy evaluation means that a term is not alway in term normal form. Terms are just as likely to be in *weak term normal form* where more processing is required to generate a term in *term normal form*. A term is in term normal for if its head has been reduced to a primitive.

A.2 G–Machine

The main idea behind the G–machine is to compile the source code into machine code. The goal is to do as much work as possible before the evaluation of the combinators by the abstract machine. In the G–machine, the source code is compiled into "G–code". Expressions in the source program are compiled into a *linear* sequence of G–code. Naive implementations of functional languages required the graph of the expression to be laboriously build and traversed at run time. This operation places considerable stress on the machine since vast amounts of time must be devoted to repeated instantiation of the body of the combinator.

The state of a G–machine is defined by the tuple $\langle C, S, H, G\rangle$ where

C	is the code sequence
S	is the stack of values (pointers into the heap)
H	is the heap
G	holds the global addresses

Lazy evaluation is achieved by overwriting the original node with an indirection to the compiled combinator body.

Updates are performed by placing an "update marker" on the S stack. Every time a function is evaluated, the node is overwritten with the result of the computation. Thus redundant computations are eliminated and future re–evaluations of this node simply return the computed value. Further, updates are the essence for lazy evaluation in the G-machine. Closures are built to suspend evaluating a code sequence. The G–machine views a closure as a piece of code along with its context. Evaluation of a suspension (ie: closure) is done by "entering the closure" where the state of the machine is saved while the context and code of the closure are loaded into the machine. To achieve lazy evaluation, the "update marker" holding the address to update is added to the closure. After the code in a closure has been evaluated, the abstract machine performs an update to overwrite the root node.

A.3 TIM machine

The TIM machine is so called because it contains variations of three basic instructions.

- TAKE n
- PUSH f
- ENTER n

The instruction TAKE removes n elements from a stack and makes a frame out of it. The PUSH instruction pushes a closure onto the stack and ENTER loads arg n into the machine. A closure in the TIM machine is defined by the code/arg pair where arg is a pointer to a list ($n \geq 0$) of arguments.

The advantages of TIM are its simplicity and ease of implementation. TIM is a spineless machine, that is, TIM does not contain a spine to unwind instead, arguments are simply stacked until needed. However, this model encounter problems with lazy evaluation. Updating becomes considerable more tricky and complicated and are handled by making each closure self updating. The state of the TIM machine is defined by

(instruction sequence, frame pointer, stack, heap, code store).

The instruction sequence contains the code acting on the frame pointed to by the *frame pointer*. The stack and heap maintain computed values and the code store associates a label (code name) with the stored address.

References

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. “The Categorical Abstract Machine”. In G. goos and J. Hartmanis, editors, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer–Verlag, September 1985.

[CF92] R. Cockett and T. Fukushima. About Charity. Research Report 92/480/18, Department of Computer Science, University of Calgary, Calgary, Alberta, CANADA T2N-1N4, June 1992.

[Dil88] A. Diller. *Compiling Functional Languages*. John Wiley & Sons LTD, 1988.

[Fai86] J. Fairbairn. A Simple Abstract Machine to Execute Supercombinators. In J. Fasel and R. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 49–52. Springer–Verlag, September–October 1986.

[Fuk91] T. Fukushima. Charity User Manual. Draft, Department of Computer Science, University of Calgary, Calgary, Alberta, CANADA T2N-1N4, November 24 1991.

[FW87] J. Fairbairn and S. Wray. “TIM: A simple, lazy abstract machine to execute supercombinators”. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer–Verlag, September 1987.

[Gor88] M. J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.

[Han91] J. Hannan. “Making Abstract Machine Less Abstract”. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 618–635. Springer–Verlag, August 1991.

[Her92] Mike Hermann. A lazy graph reduction machine for charity: Charity abstract reduction machine (CHARM). Research report, Department of Computer Science, University of Calgary, Calgary, Alberta, CANADA T2N-1N4, July 4 1992.

[Joh86] T. Johnsson. “target code generation from G–machine code”. In J. Fasel and R. Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 119–159. Springer–Verlag, September–August 1986.

[Jon92] S. L. Peyton Jones. “Implementing lazy functional languages on stock hardware: the Spineless Tagless G–machine”. In et al J. Hughes, P. Hudak, editor, *Journal of Functional Programming*, volume 2, pages 127–202. Cambridge University Press, April 1992.

[Pey] PeytonJones. Implementation tutorial.

[Wal91] R. F. C. Walters. *Categories and Computer Science*. Number 2 in Undergraduate Lecture Notes in Mathematics. Carslaw Publications, Sydney, Australia, 1991.