THE UNIVERSITY OF CALGARY

Implementing the Charity Abstract Machine

by

Dale Barry Yee

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
SEPTEMBER, 1995

© Dale Barry Yee 1995

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Implementing the Charity Abstract Machine" submitted by Dale Barry Yee in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. R. Cockett
Department of Computer Science

Dr. K. Loose
Department of Computer Science

Dr. C. Laflamme
Department of Mathematics & Statistics

Date

Abstract

This thesis describes a series of three abstract machines, with associated compilation procedures, for the Charity programming language. Each machine is a refinement of the previous, getting closer to the level of the physical machine.

The current C implementation of Charity is based on the last of these machines and is roughly twenty times faster than the original SML implementation, which was based on the first.

Acknowledgements

I would like to express my sincere gratitude to Dr. Cockett for his unlimited support, encouragement, and patience as he guided me through the process of researching and completing this thesis. It has been a rewarding experience working with Dr. Cockett and learning from him.

Special thanks to the Charity group: Tom Fukushima, Marc Schroeder, Charles Tuckey, and Peter Vesely. Their participation in the Charity project was fundamental to the success of this thesis.

In addition, I am grateful for the support and encouragement of friends in the department, especially: Ulrich Hensel, Ying Liu, Camille Sinanan, and David Spooner.

To my family: thank you for your understanding and patience. It enabled me to persevere and complete my thesis.

Contents

List of Tables

List of Figures

Chapter 1
Introduction

Functional programming languages promote a style of writing programs where small functions are glued together through composition and function application to build larger programs. This idea of building by composing is not new, as it is encouraged through structured programming in imperative languages. However, unlike these imperative languages, functional languages do not generally allow manipulation of the global state of the system through commands. Rather, the ability to modify the state has been removed, eliminating "side effects" associated with unexpected state manipulation. This makes functions "referentially transparent". A language is referentially transparent if, whenever a function is given the same arguments, it will provide the same answer. In imperative languages the presence of state (eg: global variables) destroys this transparency. Lack of referential transparency can complicate the interaction between modules, sometimes resulting in unexpected behavior.

There has been much support and argumentation for using functional programming languages [Bac78, Hug89] because of advantages such as:

- implicit memory management
- polymorphic datatypes and higher order functions
- referential transparency
- suitability for implementation on parallel machines
- shorter and more abstract programs
- mathematical foundations and formal semantics

- pattern matching and expressive syntax

Despite these advantages, functional programming languages are still widely considered to be of academic and/or educational interest only. Traditional imperative languages mirror more closely machine architecture and thus are able to produce programs which are more resource efficient when compared with their functional versions. In fact, past measurements of central processor usage, memory usage and response times of imperative programs illustrate the superior resource usage over functional language counterparts [Pau91].

Considerable study has been devoted to designing evaluators and compilers which close the efficiency gap between functional languages and imperative languages. Indeed, there is still considerably more research to be done before it will be possible to implement a functional language whose performance is comparable to traditional imperative languages.

Compilers for functional languages often generate code which is executed by an *abstract machine*. These abstract machines simulate the basic operation of physical machines in software. Choosing the basic operations, the strategy for evaluating the operations, and the compilation to the operations are central issues in the implementation of functional languages.

Charity is a categorical language. This means its foundations are in category theory, although it is similar in programming style to functional languages. Thus, Charity faces very much the same implementation issues as conventional functional languages. This thesis will explore the issues involved in implementing Charity by showing its evolution from a basic high level Charity abstract machine to lower level "linear" machine that closely mirrors the operation of a physical machine.

1.1. Structure of the Thesis

Chapter 2 surveys general background and related research in the implementation of existing functional languages. Next, chapter 3 introduces the Charity programming language and explores its main features through programming examples. Chapter 4 describes a very basic Charity abstract machine containing only a few operations. The addition of datatypes greatly expands the power of the language, and chapter 5 shows how datatypes are incorporated. Chapter 6 shows the implementation of the "byte coded" Charity abstract machine where the code compiled for this machine is executed in software. Optimizations to stream line the "byte coded" machine are introduced in chapter 7. Finally, chapter 8 presents the results of the implementation and discusses possible future work.

Chapter 2
Background

Traditionally, functional languages have been translated to the λ–calculus before further translation to a lower level language. The SECD machine was one of the first abstract machines to implement the λ–calculus. More importantly, the SECD machine provided a standard for specifying and reasoning about the design of an abstract machine using state transition rules. This chapter describes the λ–calculus and some of its more common evaluation techniques. Several abstract machines which implement these techniques are discussed in detail. Beginning with the SECD machine and leading to more recent developments such as the Categorical Abstract Machine (CAM) and the Spineless Tagless G–machine, the evaluation techniques presented here contribute to the development of the Charity abstract machine described in the rest of this thesis.

2.1. The λ–calculus

When implementing a functional language, it is practical to separate the high level programming language, with all its syntactic embellishments, from the low level language. An intermediate representation serves as an interface between the two levels, making high level syntax and low level implementation of the language independent of each other. By far, the most common intermediate representation for functional programming languages is the λ–calculus. The λ–calculus provides a notation for functions where all of the "syntactic sugar" associated with programs written at a higher level is removed, resulting in a succinct notation with four basic syntactic constructs:

$$\begin{array}{rcll} exp & ::= & C & \text{constant} \\ & & v & \text{variable} \\ & & exp\ exp & \text{application} \\ & & \lambda v.exp & \text{abstraction} \end{array}$$

Examples of λ–expressions

8	a constant
$+$	a constant primitive function
x	a variable
$\lambda x.2 + x$	an abstraction (add 2 to a number)
$(\lambda x.2 + x)\ 3$	an application

Constants are not needed in the λ–calculus, as the other three constructs can be used to represent them, but it is more convenient to include them in the language.

2.2. Evaluation techniques

Evaluation of a λ–expression is done either by evaluating a constant (eg: the constant primitive function +) or through the conversion rules, the fundamental one being β–reduction. β–reduction will substitute all occurrences of a variable in an abstraction with a value. For example,

$$\begin{array}{rcll} (\lambda x.2 + x)\ 3 & \rightarrow & 2 + 3 & \text{substitution} \\ & \rightarrow & 5 & \text{evaluation of } + \end{array}$$

Evaluating either a constant or an application via β–reduction is called a reduction. A single reduction of a λ–expression is denoted by $\rightarrow$, and when no more reductions are possible, the expression is said to be in reduced or normal form. Evaluation of an expression is done through an iteration of reductions until normal form is reached. The three main reduction strategies for evaluating λ–expressions are by–value, by–

name, and by–need evaluation.

by–value evaluation

The by–value strategy (also known as eager evaluation) evaluates all arguments to a function before the function is evaluated. In the expression

$$(\lambda x.(\lambda y.y + x)\ 2)\ ((\lambda z.z + 3)\ 3)$$

a by–value strategy would reduce the expression by replacing the z with 3 first:

$$\begin{aligned}
(\lambda x.(\lambda y.y + x)\ 2)\ ((\lambda z.z + 3)\ 3) &\rightarrow (\lambda x.(\lambda y.y + x)\ 2)\ (3 + 3) \\
&\rightarrow (\lambda x.(\lambda y.y + x)\ 2)\ 6 \\
&\rightarrow (\lambda x.2 + x)\ 6 \\
&\rightarrow 2 + 6 \\
&\rightarrow 8
\end{aligned}$$

by–name evaluation

A by–name strategy replaces the variables in a function with the expressions to which they refer. That is, a function does not evaluate its argument before substitution, but rather substitutes the body of the argument. Only when a function requires the value of its argument does it actually evaluate it. A by–name strategy would reduce the example in the previous section as follows:

$$\begin{aligned}
(\lambda x.(\lambda y.y + x)\ 2)\ (\lambda z.z + 3) &\rightarrow (\lambda y.y + (\lambda z.z + 3))\ 2) \\
&\rightarrow 2 + ((\lambda z.z + 3)\ 3) \\
&\rightarrow 2 + (3 + 3) \\
&\rightarrow 2 + 6 \\
&\rightarrow 8
\end{aligned}$$

A drawback of this evaluation strategy is when a variable occurs multiple times in a function. For example,

$$(\lambda x.x + x)\ (2 + 3)$$

would require the argument $2+3$ to be evaluated twice, as in the following reduction:

$$\begin{array}{rcl}
(\lambda x.x+x)\ (2+3) & \rightarrow & (2+3)+(2+3) \\
 & \rightarrow & 5+(2+3) \\
 & \rightarrow & 5+5 \\
 & \rightarrow & 10
\end{array}$$

by–need evaluation

A by–need evaluation strategy (also referred to as lazy evaluation) is a by–name evaluation strategy which *shares* the values of arguments. That is, arguments to the function are evaluated at most once. The evaluated argument shares its result with the rest of the expression. For example, in the reduction below, the value of f is shown on the right:

$$\begin{array}{rcl|r}
\text{let } f & = & (2+3) \text{ in} & \\
 & & (\lambda x.x+x)\ f & f=(2+3) \\
 & \rightarrow & f+f & f=(2+3) \\
 & \rightarrow & (2+3)+f & f=(2+3) \\
 & \rightarrow & 5+f & f=5 \\
 & \rightarrow & 5+5 & f=5 \\
 & \rightarrow & 10 & f=5
\end{array}$$

Notice that the syntax has been extended to add a let statement which binds the name f to the expression $(2+3)$. Several steps in the reduction have been saved since the value of f is only computed once and the second reference to f will return the precomputed answer.

The choice of evaluation order has implications for the number of reductions that must be performed. In the extreme case, the argument to a function contains an expression which reduces infinitely. This causes the entire expression never to terminate under a by–value strategy:

$$\begin{aligned}(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz)) &\rightarrow (\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))\\ &\rightarrow (\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))\\ &\rightarrow \ldots\end{aligned}$$

A by–need evaluation of the above expression is:

$$(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz)) \rightarrow \lambda y.y$$

where after one reduction, the expression is in normal form. The first argument to the leftmost lambda expression has been discarded, eliminating the infinite reduction. A by–need evaluation strategy will always have no more (and usually less) reductions than a by–value strategy. However there is an overhead as evaluation of arguments must be suspended. This requires extra storage for the suspensions, or “closures”. In all casees where the reduction of an expression terminates, both the by–value and the by–need strategy will reduce to the same result as the reduction system is confluent (Church–Rosser [FH88]). Also, if the normal form exists, the by–need strategy is guaranteed to find it.

2.3. Implementing functional languages

Often a functional language is implemented by defining an abstract machine that details how the evaluation of an expression is to proceed. This abstract machine replaces the real machine for the purpose of formally modelling, simulating, optimizing, and reasoning about an evaluation technique. An abstract machine allows the designer to concentrate on defining how the state of the system evolves to another state through state transition rules. This facilitates reasoning about the design of the basic operations at an abstract, formal level.

This section introduces several different approaches to building abstract machines. An environment based SECD machine is presented, followed by several graph reducing machines. An environment based implementation associates variables with their values on a "lookup stack". Graph reduction schemes use directed graphs with a set of graph reduction rules to reduce the graph to a normal form. Graph reduction implements by–need (or lazy) reduction, where a portion of a graph is overwritten with an equivalent simpler graph so that subsequent visits to that portion of the graph will simply return an answer in normal form. The ease with which shared subexpressions are represented graphically provides a more intuitive approach to by–need evaluation than environment based evaluation strategies do.

2.3.1. SECD machine

The SECD machine [Lan64] provides an implementation of the λ–calculus in a mechanical and straightforward manner. The abstract machine for Charity shall use the same format for describing the state transitions.

The SECD machine has four stacks which determine the state of the machine. These stacks are:

- S stack, stack of intermediate values
- E stack, the environment used for variable lookup
- C stack, the code stream
- D stack, the dump (a stack of stacks)

State transitions define how the SECD machine is to evolve from one state to the next. The current state is given by the contents of each of the four stacks at any given time and transitions are denoted:

$$S \quad E \quad C \quad D \;\rightarrow\; S' \quad E' \quad C' \quad D'$$

where the state of the machine matching the left side of the arrow is transformed to a new state on the right side of the arrow.

The SECD machine provided an implementation of the λ–calculus where application of expression as well as abstraction and substitution of variables are implemented.

The state transitions of the SECD machine as presented in [FH88] follow. State transition 1 stops the execution in the machine when the code stack and dump stack are empty. State transition 2 loads the machine with the stacks stored after an application of a function to its argument has been completed. The notation . is used to push an item onto the top of a stack (eg: $2.s$ is stack with 2 on top and then s). An empty code stream is denoted with ϵ as the C stack.

	S	E	C	D		S'	E'	C'	D'
1	$x.s$	e	ϵ	$[\,]$	$\rightarrow$		STOP		
2	$x.s$	e	ϵ	$(s', e', c').d$	$\rightarrow$	$x.s'$	e'	c'	d

In transition 3, when a variable is encountered on the code stream, a lookup into the E stack is performed to replace the variable with its actual value. The variable (or a primitive function such as +) is tagged with an ID and placed on the S stack. For λ abstractions tagged with LAM, the SECD machine stacks the current variables and arguments to the abstraction onto the S stack as shown by transition 4. A closure (tagged with CLO) holds the body of the abstraction, the variables and the environment.

	S	E	C	D		S'	E'	C'	D'
3	s	e	$\mathrm{ID}(x).c$	d	$\rightarrow$	$\mathrm{lookup}(x, e).s'$	e	c	d
4	s	e	$\mathrm{LAM}(vars, body).c$	d	$\rightarrow$	$\mathrm{CLO}(body, vars, e).s$	e	c	d

State transition 5 and 6 shows how application is handled. The application symbol '@' on the code stack causes the first two arguments on top of the stack to be applied together. In transition 5 a closure on top of the S stack causes the body of the abstraction to be executed and the variables to be bound to its value. A primitive function such as + on top of the S stack is applied to the next argument on the S stack in transition 6.

	S	E	C	D		S'	E'	C'	D'
5	$\mathrm{CLO}(body, vars, e').(arg.s)$	e	$@.c$	d	$\rightarrow$	$[\,]$	$(vars, arg).e'$	$[body]$	$(s, e, c).d$
6	$\mathrm{PRIM}(f).(arg.s)$	e	$@.c$	d	$\rightarrow$	$f(arg).s$	e	c	d

Given an application (APP) of two expressions fa, transition 7 of the SECD machine first evaluates a, then f and finally the application before the application @ takes place.

	S	E	C	D		S'	E'	C'	D'
7	s	e	APP$(fun, arg).c$	d	$\rightarrow$	s	e	$arg.(fun.(@.c))$	d

As an example of how the SECD machine reduces an expression, consider the λ–expression $(\lambda x.x + 2)$ 3. The SECD code for this expression is

$$APP(LAM(ID(x), APP(ID(+), APP(ID(2), ID(x)))), ID(3))$$

where the + is in infix notation.

	S	E	C	D
	[ ]	[ ]	APP(LAM(ID(x), APP(ID(+), APP(ID(2), ID(x)))), ID(3))	[ ]
$\rightarrow$	[ ]	[ ]	ID(3).(LAM(ID(x), APP(ID(+), APP(ID(2), ID(x)))).@)	[ ]
$\rightarrow$	[3]	[ ]	LAM(ID(x), APP(ID(+), APP(ID(2), ID(x)))).@	[ ]

Encountering a LAM instruction on the code stack causes a closure to be placed on the S stack.

	S	E	C	D
$\rightarrow$	CLO(APP(ID(+), APP(ID(2), ID(x)), LAM(ID(x), [ ])).3)	[ ]	@	[ ]

Next, the application @ causes the evaluation of the closure where the variable x will be replaced with 3.

	S	E	C	D
$\rightarrow$	[ ]	$[(x, 3)]$	APP(ID(+), APP(ID(2), ID(x)))	$[([\,], \epsilon, [\,])]$
$\rightarrow$	[ ]	$[(x, 3)]$	APP(ID(2), ID(x)).(ID(+).@)	$[([\,], \epsilon, [\,])]$
$\rightarrow$	[ ]	$[(x, 3)]$	ID(x).(ID(2).(ID(+).@).@)	$[([\,], \epsilon, [\,])]$
$\rightarrow$	[3]	$[(x, 3)]$	ID(2).((ID(+).@).@)	$[([\,], \epsilon, [\,])]$
$\rightarrow$	2.3	$[(x, 3)]$	(ID(+).(@.@))	$[([\,], \epsilon, [\,])]$

Finally, the machine is in a position to evaluate the values stacked up on the S stack:

	S	E	C	D
$\rightarrow$	$+.2.3$	$[(x,3)]$	@.@	$[([\,],\epsilon,[\,])]$
$\rightarrow$	$(+\ 2).3$	$[(x,3)]$	@	$[([\,],\epsilon,[\,])]$
$\rightarrow$	5	$[(x,3)]$	$[\,]$	$[([\,],\epsilon,[\,])]$
$\rightarrow$	5	$[\,]$	ϵ	$[\,]$

The SECD machine described above executes in a by–value manner, since an argument is evaluated before the expression is evaluated. There are also by–need variants of the SECD machine that evaluate the function before the arguments. These by–need machines actually suspend the evaluation of an argument until the argument is needed.

Turner [Tur79] observed that the SECD machine spends considerable time looking up variables and their actual values from the E stack to perform a substitution. The combinator based machines described in the next section solve this problem by eliminating free variables from an expression.

2.3.2. Combinators

One of the problems with intermediate languages such as the λ–calculus revolves around substituting variables properly. Substitution requires determining where free variables in the λ–expression occur and the impact of substituting another parameter into the expression. For instance, expression $(\lambda x.\lambda y.x+y)\ y\ 7$ may appear to have the following reduction:

$$\begin{aligned}(\lambda x.\lambda y.x+y)\ y\ 7 &\rightarrow (\lambda y.y+y)\ 7\\ &\rightarrow 7+7\end{aligned}$$

In this expression, the y being substituted is free, but after it is substituted for the x in the lambda abstraction, it becomes bound. The correct substitution would have produced $y+7$, if the y passed as an argument is differentiated from the y in the expression:

$$\begin{aligned}(\lambda x.\lambda z.x + z)\ y\ 7 &\rightarrow (\lambda z.y + z)\ 7\\ &\rightarrow y + 7\end{aligned}$$

During substitution of a variable in a λ–expression, locally bound variables must be renamed to avoid name clashes as shown above.

Combinators are closed lambda expressions that have no free variables. For example, $\lambda x.x$ is a combinator while $\lambda x.y$ is not, since the x in the former expression bound to λx, while the y in the later expression is unbound. Combinators eliminate free variables in a λ–expression to avoid the problem with substitution.

The Miranda system [Tur85] is implemented with the Turner set [Tur79] of combinators. This system contains three fixed combinators[1]:

$$\begin{aligned}\mathrm{S} &= \lambda f.\lambda x.\lambda\sigma.(f\sigma)(x\sigma)\\ \mathrm{K} &= \lambda x.\lambda\sigma.x\\ \mathrm{I} &= \lambda x.x\end{aligned}$$

which can represent any lambda expression. There are several translation algorithms from the λ–calculus to these combinators, each differing in the number of combinators generated. Typically, the environment σ is created as a means of storing the free variables of an expression before the expression is evaluated. The S combinator evaluates each of the f and x in the environment σ before applying f to x. The K combinator eliminates the environment. The small set of combinators is amenable to implementation on a machine, since each combinator corresponds to simple and mechanical instructions. Combinators eliminate the need to handle variable substitution and the scope of variables.

The relative ease with which combinators can be implemented makes a combinator based implementation desirable. Translation and evaluation are straightforward, furthermore the efficiency of the execution can be improved by enlarging the set of combinators. The problem of variable scope and substitution are resolved quite easily

[1] The I combinator can be defined in terms of S and K by I = S K K.

at the translation stage. In addition, the expensive lookup step of the SECD machine is avoided during execution.

Supercombinators

Hughes [Hug84] pointed out that the amount of computation performed by each combinator is relatively small. In addition, many combinators are often needed to represent a function. Hughes therefore proposed the introduction of supercombinators in order to reduce the number of combinators required to express a function. Supercombinators have a distinct advantage over Turner combinators in that the granularity of each combinator is larger since each function added to the system will create a new set of supercombinators.

In a supercombinator, free variables are eliminated from the λ expression by making all of the free variables parameters of the expression. This process of removing all free variables from a λ–expression is also known as λ–lifting. Hughes extends λ–lifting to abstract whole subexpressions containing free variables. These expressions are then used as arguments to the original expression.

The example below shows how a free variable can be eliminated by translation of λ–expression to supercombinators. In the original λ–expression:

$$f\ x = (g\ (\lambda y.x \times x + y)\ 3) + (g\ (\lambda y.x \times x + y)\ 4)$$

the variable x is free in both λ–abstractions.

Applying the λ–lifting algorithm (chapter 6 of [Jon]) would create a new supercombinator g' and redefine f as follows:

$$\begin{aligned} g'\ x\ y &= \lambda y.x \times x + y \\ f\ x &= (g'\ x\ 3) + (g'\ x\ 4) \end{aligned}$$

The supercombinator g' holds the entire expression where the previously free variable x is now bound to the extra parameter x. The supercombinator f no longer contains any free variables.

The Categorical Abstract Machine (CAM)

The Categorical Abstract Machine (CAM) is an abstract machine based on combinators derived from category theory: they include the composition and identity combinators. To this system, products and exponentials are added, giving the basic rewrite rules of the CAM machine. The evaluation of these combinators can easily be transformed to machine instructions.

The equations below provide basic equations from category theory (juxtaposition is application).

$$\begin{array}{llcl}
(Ass) & (x \circ y)\ z & = & x\ (y\ z) \\
(Fst) & Fst\ < x, y > & = & x \\
(Snd) & Snd\ < x, y > & = & y \\
(dpair) & < x, y >\ z & = & < x\ z, y\ z > \\
(d\Lambda) & \Lambda(x)\ y\ z & = & x\ (y, z) \\
(app) & App\ (x, y) & = & x\ y \\
(quote) & ('x)\ y & & x
\end{array}$$

The (Ass) gives the associative equations. The equations (Fst) and (Snd) are the projections on a pair, while $(dpair)$ is the distributive rule. Currying is accomplished by the $(d\Lambda)$ while an application is defined by the equation (app). The $(quote)$ is a constant.

The state of the abstract machine is a triple (V, C, D) with state transitions in the form:

$$(V, C, D) \Longrightarrow (V', C', D')$$

The state transitions of this machine derived from the above equations are:

V	C	D		V'	C'	D'
$\langle v_0, v_1\rangle$	$fst.c$	d	$\Longrightarrow$	v_0	c	d
$\langle v_0, v_1\rangle$	$snd.c$	d	$\Longrightarrow$	v_1	c	d
v	$'k.c$	d	$\Longrightarrow$	k	c	d
v	$\Lambda(c').c$	d	$\Longrightarrow$	$(c'\ v)$	c	d
v	$\langle.c$	d	$\Longrightarrow$	v	c	$v.d$
v_0	$,.c$	$v.d$	$\Longrightarrow$	v	c	$v_0.d$
v_1	$\rangle.c$	$v_0.d$	$\Longrightarrow$	$\langle v_0, v_1\rangle$	c	d
$\langle k.\ v_0, v_1\rangle$	$app.c$	d	$\Longrightarrow$	$\langle v_0, v_1\rangle$	d	
v	ϵ	d	$\Longrightarrow$	STOP		

The ($dpair$) combinator distributes a value over a pair and is split into three operations in the CAM machine. The “$\langle$” pushes the current value v on the dump, while the “,” swaps the value of the dump with the value stack. Lastly, the “$\rangle$” creates a pair with the value stack and top of the dump stack. For example, if the current configuration of the machine is:

V	C	D
v	$\langle c_0, c_1\rangle$	d

where $c_0 : v \to v_0$ and $c_1 : v \to v_1$. The machine would execute from this initial state as follows:

V	C	D
v	$\langle c_0, c_1\rangle$	d
v	$c_0, c_1\rangle$	$v.d$
v_0	$, c_1\rangle$	$v.d$
v	$c_1\rangle$	$v_0.d$
v_1	$\rangle$	$v_0.d$
$\langle v_0, v_1\rangle$	ϵ	d

In the development of the Charity abstract machine in Chapters 5 to 7 we adapt the CAM machine to evaluate a different set of categorical combinators. The above execution of the implementation of the pair combinator is particularly interesting since we will incorporate the same evaluation technique on products in chapter 7.

2.3.3. Other implementations

Compiled implementations attempt to reconcile the vastly different approaches of graph reducers, stack based, and combinator implementations with current compiler technology for imperative languages. To implement functional languages on current machines, compiled implementations borrow heavily from their imperative counterparts, while trying to retain the optimizations and formal framework of the traditional functional paradigm.

Spineless Tagless G–machine

The spineless tagless G–machine [Jon92] is a synthesis of the G–machine [Kie85] and the TIM [FW87] machine, borrowing the best of both architectures. The aim of this machine is to unify many of the principles of traditional compiler technology with those of functional language compilers. The discussion of the spineless tagless G–machine begins by introducing the G–machine and TIM machine.

The main idea behind the G–machine is to provide a compiled implementation of some term logic into G–code. This G–code performs a linear sequence of operations to construct, traverse and reduce the graph in the heap. The process of compiling flattens the tree structure of the expression into G–code in a postfix form. That is, the children of an expression are evaluated before a node. A simple example would be the postfix evaluation of arithmetic expressions. Lazy graph reduction in this machine is achieved by overwriting (updating) the root node after the expression has been evaluated.

The TIM machine contains only three instructions: take, push and enter. These three instructions build, traverse and evaluate the supercombinator expression represented as a graph. Compiling the term logic to TIM machine instructions is a process of flattening as in the G–machine and tupling shared expressions so that common subexpressions are only evaluated once.

The push instruction pushes an argument onto the stack. When there are enough arguments on the stack to evaluate the expression, a take instruction takes these arguments and builds a frame. Arguments in a frame are accessed by entering it. The TIM machine is a spineless machine in that there is no spine to traverse. The spine of an expression refers to the unwinding of an expression (graph) to find the next reduction. The TIM does not require any unwinding since the compiled instructions build and reduce the graph.

As the name implies, the spineless tagless G–machine is spineless in the sense of the TIM machine. However, this machine is also tagless because objects in this machine point to actual code instead of using a tag to differentiate two objects. Similar to the G–machine, objects in this system point to the actual code to execute, avoiding the cost of interpreting some instruction.

The implementation of the Charity abstract machine will use the techniques of lazy graph reduction to evaluate combinators compiled to "macro code". Thus, many techniques introduced in this section contribute to the development of the Charity abstract machine. For example, the closure mechanism of the TIM machine is incorporated into the evaluation of coinductive combinators. The evaluation of these combinators are suspended until their evaluation is forced by applying a destructor (more on the implementation can be found in chapter 6).

Chapter 3

The Charity programming language

This chapter describes the programming language Charity. While the language has foundations in category theory (see [CF92]), one does not require any knowledge of it to write programs. Charity programs resemble those written in other functional languages, where a collection of simple functions are combined together to produce a program. In addition, features such as strong type checking, which resolve the types of all functions at compile time, ensure that composed functions are compatible. This type checking mechanism guarantees type mismatch errors can never arise at run time.

User defined polymorphic datatypes are the building blocks of Charity programs. A central feature of Charity is the separation of the datatypes into two classes: inductive and coinductive. Inductive datatypes include many familiar data structures used in computer science, such as a lists and trees. The pure inductive datatypes are finite. Coinductive datatypes such as infinite lists are the dual of the inductive datatypes. Coinductive datatypes such as colists and cotrees are potentially, but not necessarily, infinite in nature.

The Charity language has a very simple syntax, with only six basic operations for data manipulation, as well as a means of defining both datatypes and functions. Defining a datatype in this system generates three of the six operations for manipulating data. The inductive datatypes produce the operations case, fold and map^L while the coinductive datatypes yield the operations record, unfold and map^R.

Charity programs are strongly normalizing, in the sense that terms will always reduce to a normal form after a finite series of reductions. General recursion, which sometimes has been called the "goto of functional languages", is not permitted in this language. Thus, the possibility of writing infinitely recursive functions is eliminated. Charity promotes a disciplined style of programming where data can only be accessed and manipulated through one of three operations delivered by datatype declarations.

Charity programs are expressed in a term logic which will be described through examples in the remainder of this chapter.

3.1. Basic types and terms

There are two basic types in Charity:

- 1 (unit)
- × (products).

These two basic types come equipped with term formation rules. The unit type has the term formation rule:

$$\frac{}{() : 1}$$

which allows introduction of a basic term (the 0–tuple). Given the types (or terms) on the top line, the term on the bottom line can be inferred. The above rule says that "from nothing, a term () can be produced which has type 1".

The term formation rules for products describe how pairs and the projections are created:

$$\frac{a : A \quad b : B}{(a, b) : A \times B} \qquad \frac{x : A \times B}{P_0(x) : A} \qquad \frac{x : A \times B}{P_1(x) : B}$$

In addition, Charity provides a means of abstracting terms similar to the λ abstraction of chapter 2. Charity abstractions are written as:

```
{ v => t }
```

and have the following term formation rule:

$$\frac{v \in A \quad t : B}{\{v \mapsto t\} :: A \to B}$$

where v is a variable base (a sequence of variables, see appendix A) and t is the term being abstracted. Note that abstractions are not terms. Rather, they are first order functions as indicated by the (::).

Evaluation of functions is done with the rule:

$$\frac{f :: A \to B \quad t : A}{f(t) : B}$$

which, given a function f and a term t, forms a new term $f(t)$.

3.2. Inductive datatypes

Inductive datatypes provide the classical data structures of computer science. The syntax for inductive datatype definitions is:

$$\begin{array}{rlll} \text{data } L(A) \longrightarrow S = & c_1 : E_1(A,S) & \longrightarrow & S \\ \mid & \vdots & & \\ \mid & c_n : E_n(A,S) & \longrightarrow & S. \end{array}$$

This definition introduces a new type L and constructors c_i $(i = 1...n)$. The type L is parametric about a type variable A (which stands for a finite sequence of types). The constructors c_1 through c_n are used in building elements of type $L(A)$. They have types:

$$c_i : E_i(A, L(A)) \to L(A)$$

where the state variable S has been replace with $L(A)$.

3.2.1. Booleans

One of the most basic datatypes is the boolean type. Boolean and its two constructors, false and true, are delivered to the system by the declaration:

```
data bool -> C = false: 1 -> C
               | true : 1 -> C.
```

In this definition, the domain type $E(\_, C)$ equals the basic type 1 for both constructors.

A function is defined in charity using the `def` keyword. These definitions have the form:

```
def f(v) = t.
```

where `f` is a function name, `v` is a variable base, and `t` is a term. Defining a function really defines an abstraction with a label since:

```
def f = { v => t } ≡ def f(v) = t
```

Some common functions using the boolean type are the boolean `and`, `or`, and `not` defined in Charity as:

```
def and(x,y) =            def or(x,y) =             def not(x)
   { true()  => y            { true()  => y            { true()  => false
   | false() => false        | false() => false        | false() => true
   } (x).                    } (x).                    } (x).
```

These three functions introduce the case operation, the first of the three operations used in manipulating the values of inductive datatypes. The case operation examines the "root" of the data element to determine which action to perform. The general form of the case construct is:

$$\left\{\begin{array}{ccc} c_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ c_n(v_n) & \mapsto & t_n \end{array}\right\}(t)$$

where c_i is a constructor of a data type, v_i is the parameter of the constructor (a variable base, see appendix A), t_i is the phrase to evaluate when the corresponding constructor is encountered and t is the term being cased over. In the `and` example, a case over `x` determines whether `x` is `true` or `false`. In the case of `true`, the value `y` is returned, otherwise, in the case of `false`, a `false` value is returned.

In general, the case term formation rule delivered to the system upon declaration of a datatype L is:

$$\frac{\{e_i \mapsto t_i\} :: E_i(A, L(A)) \to C}{\{c_i(e_i) \mapsto t_i\}_{i=1}^{n} :: L(A) \to C}$$

Given a phrase for each constructor of datatype L, the case operator can be constructed. The function evaluation rule introduced in the last section can now be used to apply the case to a term (an element of a datatype) to produce an answer. An example of evaluation in Charity would be:

```
> and(true, false).
> false : bool
```

3.2.2. Natural numbers

The (unary) natural numbers are delivered by the following Charity declaration:

```
data nat -> C = zero: 1 -> C
              | succ: C -> C.
```

The elements of nat are `zero`, `succ(zero)`, `succ(succ(zero))`, and so on.

To determine if a natural number is odd, one could start by returning a `false` as the answer when a `zero` is encountered. Each subsequent application of a `succ` would negate the answer until all `succ`'s are exhausted. For example, to see if the number `succ(succ(zero))` is odd, using the above algorithm, one would replace `zero` by `false` resulting in `succ(succ(false))`. Upon seeing a `succ` the answer it holds would be inverted arriving at `succ(true)`. Finally, the last `succ` inverts the answer one more time producing the result of `false`.

Transforming a natural number in this manner is done through the fold operation. However, the fold operation does not start at the "leaf" of the datatype, or the `zero` when processing the natural numbers. Instead, the root is examined first, and based on the data definition, an appropriate action is taken. In the example above, the number `succ(succ(zero))`, the `succ` is encountered first. `succ` is defined recursively, meaning the value of its argument must be examined first before inverting the answer. Eventually, the `zero` constructor with a type `1` (no parameters) will return a `false`, allowing the algorithm to continue as before.

In general, the fold operation contains two stages, "top down" and "bottom up". The "top down" stage traverses through the entire data structure searching for the leaf nodes. When the leaf nodes are found, the "bottom up" stage commences. Processing occurs at each node, returning up towards the root.

The definition of the `odd` function, using the fold operation is:

```
def odd(n) =
     {| zero: ()  => false
      | succ: ans => not(ans)
      |} (n).
```

To see if a number is even, one could first see if the number is odd, then invert this answer.

```
def even(n) = not(odd(n)).
```

To add two natural numbers together, the `zero` constructor of the first number is replaced with the second number:

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

As shown by the example above, the fold construct is used to manipulate both the structure and elements of a datatype. The syntax for the fold is:

$$\left\{\left|\begin{array}{ccc} c_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ c_n : v_n & \mapsto & t_n \end{array}\right|\right\}(t)$$

where as in the case construct, each constructor c_i of a data type with parameter v_i will execute the corresponding t_i when the fold is applied to an element t. The term formation rule describing the fold says that "given maps for each constructor of the type L to some other type C, the fold operator can be inferred from the combinator of these maps". The term formation rule for the fold is shown below:

$$\frac{\{e_i \mapsto t_i\} :: E_i(A, C) \to C}{\{|c_i : e_i \mapsto t_i|\}_{i=1}^{n} :: L(A) \to C}$$

3.2.3. Lists

Lists are defined in Charity by the datatype definition:

```
data list(A) -> C = nil  : 1     -> C
                  | cons: A * C -> C.
```

The datatype `list` and its two constructors, `nil` and `cons`, are delivered to the system by this definition. Elements of this datatype are generated by constructors:

$$\begin{array}{rl} \text{nil}: & 1 \to \text{list(A)} \\ \text{cons}: & \text{A} \times \text{list(A)} \to \text{list}(A) \end{array}$$

The `nil` constructor has no parameters, as indicated by its domain type `1` (unit), while the `cons` constructor has a parameter `A` in its domain. Thus, an element of `list` would look like `cons(`a_0`, cons(`a_1`, ...,nil)))`. This is represented by the short hand notation $[a_0, a_1, ...]$.

An example of casing over lists is the function `isEmpty` which tests whether a list is empty:

```
def isEmpty(l) =
    { nil ()       => true
    | cons(a, l') => false
    } (l).
```

Note that in the above example, the case operator that transforms an element of the `list` datatype to an element of the `bool` datatype has type

$$isEmpty : list(A) \rightarrow bool$$

and each phrase corresponding its constructor has type:

$$\begin{aligned} nil &: 1 \rightarrow bool \\ cons &: A \times list(A) \rightarrow bool \end{aligned}$$

To calculate the length of a list, one replaces the `nil` in a list with the `zero` constructor and all occurrences of `cons` with `succ`. Thus, in Charity, the length function is defined as:

```
def length(l) =
    {| nil:  ()     => zero
     | cons: (a, n) => succ(n)
     |} (l).
```

The input `cons(`a_0`, cons(`a_1`, nil))` (ie: `[`a_0`, `a_1`]`) would produce `succ(succ(zero))` when the `length` function is applied to it.

Suppose one wants to increment each number in a list of natural numbers ([5, 2, 7]) by one. This can be done through the fold construct by:

```
def addOne(l) =
    {| nil: ()       => [ ]
     | cons:(x, l') => cons(succ(x), l')
     |} (l).
```

Starting with the empty list, each element of the list is applied with `succ` and `consed` onto the front of the result list. Since it is common to retain the structure of the original datatype and change the parametric variable (or the elements of a value of the datatype), Charity provides a *map* construct to accomplish this feat. The syntax for the map construct is:

$$L \left\{ \begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array} \right\} (t)$$

where m is the number of parametric type variables. An example of the map construct would be to apply the function f to each item in a list (eg: $map\ f\ [a_0, a_1, ...] = [f(a_0), f(a_1), ...]$). This map operation would be written in Charity as:

```
list{f}([a0, a1, ...]).
```

In the case of the `addOne` example above, this function can be rewritten with the map construct as:

```
def addOne'(L) = list{succ}(L).
```

The corresponding term formation rule for the map operation is:

$$\frac{f_i :: A_i \to A'_i}{L\{f_i\}_{i=1}^{m} :: L(A_1, ..., A_m) \to L(A'_1, ..., A'_m)}$$

which says that "if the maps to transform each parametric type in L are given, then the map over L can be inferred".

Note that the `bool` and `nat` datatypes do not have parametric types and have a map operation which requires no arguments and so is the trivial identity function!

3.2.4. Success or Fail datatype

The success or fail datatype is useful in raising an "exception" within charity programs. The definition of success or fail is:

```
data sf(A) -> C = ff : 1 -> C
                | ss : A -> C.
```

The `sf` datatype is often used to return answers from a function which is only partially defined. For example, the head of a list is only defined for non–empty lists. If the list is empty, an error should be produced. Using the `sf` datatype, `head` of an empty list will be the constructor `ff` (fail), while the `head` of a non–empty list will be the answer wrapped up in the `ss` (success) constructor. The Charity defintion of `head` is:

```
def head(L) =
    { nil ()      => ff()
    | cons(a, _) => ss(a)
    } (L).
```

3.3. Coinductive datatypes

Coinductive types provides a means of representing potentially infinite structures. The general form of the coinductive datatype is similar to the inductive datatype with the exception that the state variable should line up on the left side of the arrow:

$$\begin{array}{rcll} \text{data } S \longrightarrow R(A) & = & d_1 : & S \longrightarrow E_1(A,S) \\ & | & & \vdots \\ & | & d_n : & S \longrightarrow E_n(A,S). \end{array}$$

This definition delivers to the system the type $R(A)$ and destructors d_i, $i = 1, ..., n$. An element of datatype $R(A)$ is broken down by destructors with type:

$$d_i : R(A) \rightarrow E_i(A, R(A))$$

3.3.1. Infinite lists

In Charity the infinite list datatype is delivered by:

```
data S -> inflist(A) = head: S -> A
                     | tail: S -> S.
```

where, given some state `S`, the current state is returned by the destructor `head(S)` and the next state by `head(tail(S))`. In contrast to inductive datatypes, coinductive datatypes deliver the unfold, record, and map operations for manipulating their values.

The unfold construct builds a data structure one state at a time in a "lazy" manner (do only the computations needed to produce the next answer) . The unfold has the form:

$$\left(\!\left| v \mapsto \begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right|\!\right)(t)$$

The term formation rule for the unfold is:

$$\frac{v \mapsto t_i :: S \to E_i(A, S)}{(\!| v \mapsto d_i : t_i |\!)_{i=1}^{n} :: S \to R(A)}$$

To produce the infinite list of natural numbers starting from 1, one would define the following Charity function;

```
def nats =
   (| S => head: S
    |      tail: succ(S)
    |) (succ(zero)).
```

which says, given the initial state `succ(zero)`, the head thread will return the current state (in the first case `succ(zero)`), while the tail thread generates the next state. Thus the natural number $\mathrm{succ}^n(\mathrm{zero})$ would be computed by $\mathrm{head}(\mathrm{tail}^{n-1}(\mathrm{nats}))$. Of course it would be impossible to generate the entire infinite list: the unfold will only produce the values demanded of it.

Records are created through the record construct written as:

$$\left(\begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right)$$

with the term formation rule:

$$\frac{t_i : E_i(A, R(A))}{(d_i : t_i) : R(A)}$$

The record construct allows elements to be added to the head of a coinductive data structure. For example, adding the number zero to the head of the nats is achieved by:

```
def zeronats = (head: zero, tail: nats).
```

The head of the infinite list of zeronats is then zero while the tail of zeronats simply produces the infinite list of nats as defined above.

To add a constant to each element of nats, one would use the map operation:

```
inflist {succ} (nats).
```

This map over the coinductive datatypes is similar to the map over the inductive datatypes where the parametric parameters are transformed, but the structure of the datatype remains the same.

3.3.2. Colists

The definition of lists could also have been given using the sf datatype as:

```
data list'(A) -> C = enlist: sf(A * C) -> C.
```

where

$$\begin{aligned}(\text{enlist}(\text{ff}())) &\equiv \text{nil}()\\ (\text{enlist}(\text{ss}(\text{a,c}))) &\equiv \text{cons}(\text{a,c})\end{aligned}$$

Colists are the dual of the inductive lists and are defined using the sf datatype:

```
data C -> colist(A) = delist: C -> sf(A * C).
```

Notice the symmetry between the domain and codomain of the inductive and coinductive definitions. This symmetry justifies the dual nature of the coinductive and the inductive datatypes.

Elements of the colist can be finite or infinite in nature. If the colist terminates, it does so by returning a constructor ff from delist. Otherwise the next element of the list along with the next state is wrapped up in the ss constructor. For example, converting a list to a colist is accomplished by unfolding on the list:

```
def list2colist(L) =
    (| state => { nil ()      => ff
                | cons(a, L') => ss(a, L')
                } (state)
     |) (L).
```

Applying `list2colist` to `[1,2,3]` would produce the colist:

```
(delist: ss(1,
            (delist: ss(2,
                        (delist(ss(3,
                                   (delist: ff())
                                   )
                               ))
                       ))
           )
)
```

3.4. Passing functions as parameters

Normally in functional languages, functions may be higher–order. However, Charity is a first order language and does not have any higher order functions. That is to say, functions are not first class citizens in Charity. Combinators may, however, be defined to pass functions (as macros) to other functions. For example, given a list of elements, one may want to *filter* out certain elements. Suppose the list l contains natural numbers ranging from 1 to 100 (eg. $[1, 2, 3, ..., 100]$). To retain only the odd numbers of l one may write the following function in Charity:

```
def filterOdds(L) =
     {| nil:  () => [ ]
      | cons: (x, L') => { true()  => cons(x, L')
                         | false() => L'
                         } (odd(x)).
      |} (L).
```

Starting with the empty list as the result, each item in the list is applied with the odd function to test if it should be kept or discarded. If the number is odd, it is `consed` onto the front of the result list.

To filter out the even numbers, a new function filterEvens would have to be defined with the same structure as filterOdds, but a different predictate (replace "even" for "odds" above). In general, when filtering out elements of the list the structure of each function is identical with the exception of the predicate which determines the element to retain or discard. Thus one would like a general definition of the `filter` function which has a functional argument, `pred`. This is achieved via the definition:

```
def filter {pred} (L) =
    {| nil:  () => [ ]
     | cons: (x, L') => { true()  => cons(x, L')
                        | false() => L'
                        } (pred(x)).
     |} (L).
```

3.5. Other examples

Many other useful datatypes can be defined in Charity. Below are some examples:

```
data tree(A) -> C = treeLeaf: 1 -> C
                  | treeNode: C * (A * C) -> C.


data bush(A) -> C = bushLeaf: 1 -> C
                  | bushNode: list(A * C) -> C.

data S -> cotree(A) = detree: S -> sf(S * (A * S))

data S -> cobush(A) = debush: S -> sf(list(A * S)).
```

Given these and other datatypes, it is possible to write programs such as quicksort (see Appendix B), Ackermann's function (found in [CF92]), pascal's triangle, towers of Hanoi, prime numbers generation (using the sieve of Eratosthenes), and many others.

Chapter 4

The Charity Abstract Machine : basics

The evaluation strategy for Charity programs is determined by the Charity abstract machine which is an adaptation of the Categorical Abstract Machine [CCM85]. The foundations of both machines lie in category theory. However they differ somewhat in their selection of rewrite rules, combinators, and the evaluation strategy. Underlying the Charity abstract machine is a typed combinator theory in which programs, expressed as combinators, are evaluated through rewrite rules. To execute a program, the Charity system must first translate the term logic to categorical combinator expressions, and then evaluate those expressions with the Charity abstract machine. A set of state transition rules derived from the rewrite rules determine the exact manner in which the combinator expression is evaluated.

This chapter will explore the combinators and rewrite rules that are used in the formulation of the state transition rules for the Charity abstract machine. Specifically, this chapter examines the basic combinators and the function passing mechanism of the Charity system.

4.1. Categorical combinators and their types

A combinator theory consists of a system of types, a set of primitive combinators, a system for building combinator expressions from the primitive combinators, and a set of identities between combinator expressions. A categorical combinator theory is then a combinator theory where the identities between combinators are delivered from the basic structures found in category theory.

4.1.1. Types

A system of types is generated by a given set of type constructors with a fixed arities for example:

$$\begin{aligned} 1 &: \Omega^0 \to \Omega \\ \mathrm{list} &: \Omega^1 \to \Omega \\ \mathrm{tree} &: \Omega^2 \to \Omega \end{aligned}$$

The arity represents the number of parameters the given type constructor accepts. These parameters provide for a polymorphic type definition. A type expression is then a term (in the free algebraic theory) generated by the type constructors, for example:

$$\mathrm{tree}(\mathrm{list}(1), \mathrm{tree}(A, \mathrm{list}(B)))$$

4.1.2. Categorical combinators

Once a type is defined in the system, a computational framework is provided in which categorical combinators manipulate the elements of a type. A categorical combinator is a map from one type to another, which also relies on certain input parameters. A categorical combinator system on a type system is generated from primitive combinators of the form:

$$c\{S_1 \to S_1', ..., S_n \to S_n'\} : S_0 \to S_0'$$

where

- c is the combinator name,
- S_i, S_i' are type expressions, and
- $S_i \to S_i'$ is a combinator type signature.

The above combinator c accepts n input combinators with type signatures $S_i \to S'_i$, to produce a combinator with type signature $S_0 \to S'_0$. We may alternatively write this as a formation rule:

$$\frac{f_1 : S_1 \to S'_1, ..., f_n : S_n \to S'_n}{c\{f_1, ..., f_n\} : S_0 \to S'_0}$$

When such a term formation rule is introduced, it is introduced parametrically. That is, we really introduce a family of rules for each, one member for each possible substitution:

$$\frac{f_1 : (S_1)_\sigma \to (S'_n)_\sigma, ..., f_n : (S_n)_\sigma \to (S'_n)_\sigma}{c\{f_1, ..., f_n\} : (S_0)_\sigma \to (S'_0)_\sigma} \text{ for any substitution } \sigma$$

We can specialize, or instantiate, a combinator with the term formation rule:

$$\frac{e : T_1 \to T_2}{(e)_\sigma : (T_1)_\sigma \to (T_2)_\sigma} \text{ specialization}$$

To test whether a combinator expression without type annotation is valid, one would use the unification algorithm on types.

Several examples of combinators are:

$$\begin{aligned}
\text{maplist}\{A \to B\} &\;:\; \text{list}(A) \to \text{list}(B) \\
\text{pair}\{A \to B, A \to C\} &\;:\; A \to \text{product}(B, C) \\
\text{foldlist}\{1 \to C, A \times C \to C\} &\;:\; \text{list}(A) \to C
\end{aligned}$$

Combinator expressions are built up using the following rules:

$$\frac{}{I : A \to A}\text{ identity}$$

$$\frac{e_1 : T_1 \to T_2 \quad e_2 : T_2 \to T_3}{e_1; e_2 : T_1 \to T_3}\text{ composition}$$

4.1.3. Equations between combinators

The equations[1] between categorical combinators provide the semantics for the system.

The identity combinator has the following inference rule:

$$\frac{e : T_1 \to T_2}{e; I = e = I; e : T_1 \to T_2}\text{ identity}$$

Composition of combinators is also associative:

$$\frac{e_1; T_1 \to T_2 \quad e_2; T_2 \to T_3 \quad e_3; T_3 \to T_4}{(e_1; e_2); e_3 = e_1; (e_2; e_3) : T_1 \to T_4}\text{ associativity}$$

Finally, if two combinator expression are equal, then applying the substitution σ to each combinator expression will produce equal expressions.

$$\frac{e_1 = e_2 : T_1 \to T_2}{(e_1)_\sigma = (e_2)_\sigma : (T_1)_\sigma \to (T_2)_\sigma}\text{ substitution congruence}$$

4.2. Combinators for products

An 0–ary product is called a final object and is denoted by the type 1. The combinator ! is a family of maps defined by:

$$\frac{}{! : T \to 1}\text{ terminal map}$$

with equations given by:

[1] We refer to identities as equations to avoid ambiguity

$$\frac{f : T \to 1}{f = \,! : T \to 1}\ \text{uniqueness}$$

The first formation rule establishes the existence of a map from every type in the system to type 1 while the second equation rule states that it is unique.

An alternative graphical view of combinators and their equations is obtained through *commuting diagrams.* The existence of a natural map (the ! combinator) from every type in the system to the final unit is shown by the following diagram:

$$\begin{tikzcd}
X \arrow[d, "c"'] \arrow[dr, "!_X"] & \\
Y \arrow[r, "!_Y"'] & 1
\end{tikzcd}$$

where the domain type Y is transformed to the codomain type 1 via the map $!_Y$. For any other type X with a map $c : X \to Y$, the map $!_X : X \to 1$ exists and $!_X = c; !_Y$. Commuting diagrams provide a graphical means of representing the action of combinators over the types and reasoning about them.

In addition to the above diagram, in order to secure the uniqueness of the ! combinator we require the following additional equality:

$$!_1 = I_1$$

Suppose now $f : X \to 1$ then

$$\begin{tikzcd}
X \arrow[rr, "f"] \arrow[dr, "!_X"'] & & 1 \arrow[dl, "!_1"] \\
& 1 &
\end{tikzcd}$$

commutes. Thus

$$!_x = f; !_1 = f; I_1 = f$$

showing uniqueness

The definition of products produces several combinators, and we introduce them by the following formation and equality rules:

$$\frac{f : C \rightarrow A \quad g : C \rightarrow B}{pair\{f, g\} : C \rightarrow A \times B}\ \text{pairing}$$

$$\frac{A\ \text{type} \quad B\ \text{type}}{p_0 : A \times B \rightarrow A \quad p_1 : A \times B \rightarrow B}\ \text{projections}$$

$$\begin{array}{c} h; p_0 = f : C \rightarrow A \quad h; p_1 : C \rightarrow B \\ \hline\hline h = pair\{f, g\} : C \rightarrow A \times B \end{array}\ \text{uniqueness}$$

Notice that the uniqueness rule is a two way rule meaning the equalities hold in both directions.

Expressing products (often expressed with the infix ×) with a commuting diagram yields:

$$\begin{tikzcd}
X & X \times Y \arrow[l, "P_0"'] \arrow[r, "P_1"] & Y \\
& C \arrow[ul, "f"] \arrow[u, dashed, "\langle f, g\rangle"] \arrow[ur, "g"'] &
\end{tikzcd}$$

where $\langle f, g\rangle$ is short hand for the $pair\{f, g\}$. The product diagram can be read as:

> "given a type $X \times Y$ and two combinators $P_0 : X \times Y \rightarrow X$, $P_1 : X \times Y \rightarrow Y$, and any type C with combinators $f : C \rightarrow X$ and $g : C \rightarrow Y$ there exists a unique combinator $\langle f, g\rangle$ such that $\langle f, g\rangle; P_0 = f$ and $\langle f, g\rangle; P_1 = g$".

From the above equations we can derive the distributive rule for composition over pairing:

$$f; \langle x, y\rangle = \langle f; x, f; y\rangle$$

since

$$\begin{aligned} f;\langle x,y\rangle &= \langle f;\langle x,y\rangle;p_0, f;\langle x,y\rangle;p_1\rangle \\ &= \langle f;x, f;y\rangle \end{aligned}$$

In the Charity system, the unit and product types are built in along with combinators:

$$\begin{aligned} !\{\} &: X \to 1 \\ P_0\{\} &: X \times Y \to X \\ P_1\{\} &: X \times Y \to Y \\ pair\{C \to X, C \to Y\} &: C \to X \times Y \end{aligned}$$

4.3. Translating to combinators

So far, we have introduced the basic types and combinators for the Charity system. We now define the translation $\mathcal{T}$ from Charity term logic to combinators. This case–based translation involves pattern matching over a term to produce an equivalent combinator expression. Below, a term matching the form on the left side of the equal sign will be translated to the combinator on the right side, recursively:

(T_1) $\mathcal{T}\left[v \mapsto ()\right] = !,$

(T_2) $\mathcal{T}\left[x \mapsto x\right] = I,$

(T_3) $\mathcal{T}\left[(v_0, v_1) \mapsto x\right] = p_i; \mathcal{T}\left[v_i \mapsto x\right]$ where $i = 0$ if x occurs in v_0, otherwise $i = 1$,

(T_4) $\mathcal{T}\left[v \mapsto p_i(t)\right] = \mathcal{T}\left[v \mapsto t\right]; p_i$ for $i = 0, 1$,

(T_5) $\mathcal{T}\left[v \mapsto (t_0, t_1)\right] = \langle \mathcal{T}\left[v \mapsto t_0\right], \mathcal{T}\left[v \mapsto t_1\right]\rangle$,

(T_6) $\mathcal{T}\left[v \mapsto \{w \mapsto t'\}(t)\right] = \langle \mathcal{T}\left[v \mapsto t\right], I\rangle; \mathcal{T}\left[(w, v) \mapsto t'\right]$

Abstractions are translated by the last translation (T_6). This must propagate an environment to the abstracted term. This is done using the second component of the pair. Thus, a variable in v is in the second component of the pair composed with the abstraction. The equivalence of the combinator expressions and the term logic, and thus the correctness of the translation from term logic to combinators, is proven in

[CS95].

To translate from term logic, one would mechanically pattern match the term with the translation rules to produce a categorical combinator expression. For example, the Charity expression

```
((x,y),z) => p0(p1(x,(y,z)))
```

has the following translation to combinators:

$$
\begin{aligned}
&\underline{\mathcal{T}\,[((x,y),z) \mapsto P_0(P_1(x,(y,z)))]}\\
&\quad\Longrightarrow_{T_4} \underline{\mathcal{T}\,[((x,y),z) \mapsto P_1(x,(y,z))]}; P_0\\
&\quad\Longrightarrow_{T_4} \underline{\mathcal{T}\,[((x,y),z) \mapsto (x,(y,z))]}; P_1; P_0\\
&\quad\Longrightarrow_{T_5} \langle \underline{\mathcal{T}\,[((x,y),z) \mapsto x]}, \mathcal{T}\,[((x,y),z) \mapsto (y,z)]\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; \underline{\mathcal{T}\,[(x,y) \mapsto x]}, \mathcal{T}\,[((x,y),z) \mapsto (y,z)]\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; P_0; \underline{\mathcal{T}\,[x \mapsto x]}, \mathcal{T}\,[((x,y),z) \mapsto (y,z)]\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; P_0; I, \underline{\mathcal{T}\,[((x,y),z) \mapsto (y,z)]}\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_5} \langle P_0; P_1; I, \langle \underline{\mathcal{T}\,[((x,y),z) \mapsto y]}, \mathcal{T}\,[((x,y),z) \mapsto z]\rangle\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; P_1; I, \langle P_1; \underline{\mathcal{T}\,[(x,y) \mapsto y]}, \mathcal{T}\,[((x,y),z) \mapsto z]\rangle\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; P_1; I, \langle P_0; P_1; \underline{\mathcal{T}\,[y \mapsto y]}, \mathcal{T}\,[((x,y),z) \mapsto z]\rangle\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_2} \langle P_0; P_1; I, \langle P_0; P_1; I, \underline{\mathcal{T}\,[((x,y),z) \mapsto z]}\rangle\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_3} \langle P_0; P_1; I, \langle P_0; P_1; I, P_1; \underline{\mathcal{T}\,[z \mapsto z]}\rangle\rangle; P_1; P_0\\
&\quad\Longrightarrow_{T_2} \langle P_0; P_1; I, \langle P_0; P_1; I, P_1; I\rangle\rangle; P_1; P_0
\end{aligned}
$$

which is correct, but clearly rather inefficient!

4.3.1. Resolving variable scope

One of the primary reasons for choosing combinators as an intermediate representation is that the evaluation need not handle variable name clashes and variable scoping problems. Access to a variable outside of the scope of a combinator is restricted through the use of an *environment*.

To illustrate how the environment is used in Charity, consider the function:

```
def f (x) =
    { y => g(x, y) } (zero).
```

which contains an abstraction whose body accepts a y and calls the function g, with x and y as input. The variable y is bound within the scope of the abstraction, but the x is not. To access the unbound x, the abstraction is passed a pair consisting of the actual argument for y (ie: zero), and the environment (the actual argument for x). The translation rules take into account the extra environment for each combinator, and select either the actual argument or environment as needed.

Consider the next Charity program and its translation which illustrates how the environment and how scoping problems are avoided.

```
def f(x) = {(x,y) => x} ({(x,y) => y} (x)).
```

Notice the variable x is used in more than one context. The translation rules determine the scope of a variable by creating an environment in which a term is evaluated. In fact, translations T_3 and T_6 work in conjunction by first building the context (a variable base) in transition T_6, and then accessing the variable in the context with transition T_3. The translation of the above function f is shown below:

$$
\begin{aligned}
&\underline{\mathcal{T}\left[x \mapsto \{(x,y) \mapsto x\}(\{(x,y) \mapsto y\}(x))\right]} \\
&\Longrightarrow_{T_6} \quad \langle \underline{\mathcal{T}\left[x \mapsto \{(x,y) \mapsto y\}(x)\right]}, I\rangle; \mathcal{T}\left[((x,y),x) \mapsto x\right] \\
&\Longrightarrow_{T_6} \quad \langle\langle \underline{\mathcal{T}\left[x \mapsto x\right]}, I\rangle; \mathcal{T}\left[((x,y),x) \mapsto y\right], I\rangle; \mathcal{T}\left[((x,y),x) \mapsto x\right] \\
&\Longrightarrow_{T_2} \quad \langle\langle I, I\rangle; \underline{\mathcal{T}\left[((x,y),x) \mapsto y\right]}, I\rangle; \mathcal{T}\left[((x,y),x) \mapsto x\right] \\
&\Longrightarrow_{T_4} \quad \langle\langle I, I\rangle; P_0; \underline{\mathcal{T}\left[(x,y) \mapsto y\right]}, I\rangle; \mathcal{T}\left[((x,y),x) \mapsto x\right] \\
&\Longrightarrow_{T_4} \quad \langle\langle I, I\rangle; P_0; P_1; \underline{\mathcal{T}\left[y \mapsto y\right]}, I\rangle; \mathcal{T}\left[((x,y),x) \mapsto x\right] \\
&\Longrightarrow_{T_2} \quad \langle\langle I, I\rangle; P_0; P_1; I, I\rangle; \underline{\mathcal{T}\left[((x,y),x) \mapsto x\right]} \\
&\Longrightarrow_{T_4} \quad \langle\langle I, I\rangle; P_0; P_1; I, I\rangle; P_0; \underline{\mathcal{T}\left[(x,y) \mapsto x\right]} \\
&\Longrightarrow_{T_4} \quad \langle\langle I, I\rangle; P_0; P_1; I, I\rangle; P_0; P_0; \underline{\mathcal{T}\left[x \mapsto x\right]} \\
&\Longrightarrow_{T_2} \quad \langle\langle I, I\rangle; P_0; P_1; I, I\rangle; P_0; P_0; I
\end{aligned}
$$

The code generated from the translation is unreadable, and is analogous to an assembly language. However, the point of the translation was to remove all variables from the resultant combinator expression, which has been successfully accomplished. The translated expression above creates a few parameter/environment pairs, then projects out the value needed.

In general, if an environment is required, a pair is constructed with the second component used as the previous environment. To access a particular environment nested down several layers, one must continuously project out the second component until the desired environment is located.

4.4. Evaluation of categorical combinators

After a type and the combinators that act on that type are declared, the system must describe the behavior as rewrite rules for each combinator. These are determined by the equations between combinators by imposing a direction on the equality. The equations for products are read as "the left side of the equal sign implies the right side of the equal sign". As a result, the rewrite rules defined for the basic combinators are:

$$\begin{array}{lrcl}
(R_1) & z;! & \Longrightarrow & ! \\
(R_2) & \langle x, y\rangle; P_0 & \Longrightarrow & x \\
(R_3) & \langle x, y\rangle; P_1 & \Longrightarrow & y \\
(R_4) & z; \langle x, y\rangle & \Longrightarrow & \langle z;x, z;y\rangle \\
(R_5) & x; I & \Longrightarrow & x \\
(R_6) & I; x & \Longrightarrow & x
\end{array}$$

Rewrite rule R_1 reiterates the point that the ! combinator will take any type to 1. Rules R_2 through R_4 describe the action of the product combinators. Specifically, R_2 and R_3 are the projections of a product while R_4 is the distributive law for products (see [Wal91] for more details). Identity equations are handled by the rewrite rules R_5 and R_6.

As an example of a reduction using the above rewrite rules, suppose $x : C \to X$,

$y : C \to Y$, and $z : C \to Z$ are combinator expressions, and $E = \langle x, \langle y, z\rangle\rangle; P_1; P_0 : C \to Y$. Then the reduction of this expression E is:

$$
\begin{aligned}
\underline{\langle x, \langle y, z\rangle\rangle; P_1}; P_0 &\Rightarrow_{R_3} \underline{\langle y, z\rangle; P_0} \\
&\Rightarrow_{R_2} y
\end{aligned}
$$

The part of the expression that is being rewritten by a rewrite rule is underlined.

Another example of evaluation of the function `f(0, (1, 2)) = 1` (from the previous section):

$$
\begin{aligned}
&\underline{\langle 0, \langle 1, 2\rangle\rangle; \langle\langle I, I\rangle; P_0; P_1; I, I\rangle}; P_0; P_0; I \\
&\quad\Longrightarrow_{R_4} \langle\underline{\langle 0, \langle 1, 2\rangle\rangle; \langle I, I\rangle}; P_0; P_1; I, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_4} \langle\langle\underline{\langle 0, \langle 1, 2\rangle\rangle; I}, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_1; I, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_5} \langle\langle\langle 0, \langle 1, 2\rangle\rangle, \underline{\langle 0, \langle 1, 2\rangle\rangle; I}\rangle; P_0; P_1; I, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_5} \langle\underline{\langle\langle 0, \langle 1, 2\rangle\rangle, \langle 0, \langle 1, 2\rangle\rangle\rangle; P_0}; P_1; I, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_2} \langle\underline{\langle 0, \langle 1, 2\rangle\rangle; P_1}; I, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_3} \langle\underline{\langle 1, 2\rangle; I}, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0; P_0; I \\
&\quad\Longrightarrow_{R_5} \underline{\langle\langle 1, 2\rangle, \langle 0, \langle 1, 2\rangle\rangle; I\rangle; P_0}; P_0; I \\
&\quad\Longrightarrow_{R_2} \underline{\langle 1, 2\rangle; P_0}; I \\
&\quad\Longrightarrow_{R_2} \underline{1; I} \\
&\quad\Longrightarrow_{R_5} 1
\end{aligned}
$$

Once again, notice that the resultant combinator expression derived from the translation is quite unreadable but is variable free. Evaluation of the expression above using the rewrite rules does produce the correct result (however inefficient the combinator expression is).

4.5. The basic Charity abstract machine

Evaluation of a combinator expression to normal form is accomplished through state transitions in the Charity abstract machine. The state of the Charity abstract machine consists of the tuple

$$(v, c, d, f)$$

where

- v is the value stack holding intermediate values of a computation,
- c is the code stack listing the remaining sequence of instructions to execute
- d is the dump stack thats holds continuations so that the abstract machine can execute instructions sequentially
- f is the function stack holding a list of functions passed through definitions.

Computation begins at an initial state where the value, dump, and function stacks are empty, and the code stack contains the combinator expression. When the machine has completed computation (ie. reached normal form), the value stack holds the result while both the dump and code stacks are empty. A transition in the machine is written as:

$$v \quad c \quad d|f \quad \rightarrow \quad v' \quad c' \quad d'|f'$$

where the "before state" of the machine is given by v, c, d, and f, while the "after state" is given by v', c', d' and f'. State transition rules examine the contents of the "before states" and describe how (deterministically, in our abstract machine) the system is to evolve to the next state of the machine. For most of the state transition rules, the function stack is not utilized and the listing for this stack is omitted from the tables.

The state transitions for the primitive combinators listed below are derived directly from their rewrite rules:

	v	c	$d\|f$		v'	c'	$d'\|f'$
1	v	$!.c$	d	$\rightarrow$	$!$	c	d
2	$\langle v_0, v_1\rangle$	$P_0.c$	d	$\rightarrow$	v_0	c	d
3	$\langle v_0, v_1\rangle$	$P_1.c$	d	$\rightarrow$	v_1	c	d
4	v	$\langle c_0, c_1\rangle.c$	d	$\rightarrow$	v	c_0	$\mathrm{pr}_0(v, c_1, c).d$
5	v_0	ϵ	$\mathrm{pr}_0(v, c_1, c).d$	$\rightarrow$	v	c_1	$\mathrm{pr}_1(v_0, c).d$
6	v_1	ϵ	$\mathrm{pr}_1(v_0, c).d$	$\rightarrow$	$\langle v_0, v_1\rangle$	c	d
7	v	ϵ	$\mathrm{cont}(c).d$	$\rightarrow$	v	c	d
8	v	ϵ	$[\,]$	$\rightarrow$		STOP	

Transitions 1 to 3 correspond closely to the rewrite rules. Transition 1 is the terminal combinator, and provides a starting point for all computations. Transitions 2 and 3 are the projections for products. To implement the distributive rule, transition 4 saves c_1 and a copy of v on the dump, then executes c_0 with v. Once a result v_0 has been computed, v is composed with c_1 (transition 5), resulting in v_1. Finally, the pair $\langle v_0, v_1\rangle$ is formed and left on the v stack. Transition 7 allows the machine to continue computation from some earlier suspension and transition 8 indicates that computation is complete, and that the value is in normal form.

An example of reducing the combinator expression $\langle x, \langle y, z\rangle\rangle.P_1.P_0$ to normal form using the abstract machine follows:

	v	c	d
	$[\,]$	$\langle x, \langle y, z\rangle\rangle.P_1.P_0$	$[\,]$
$\Longrightarrow_4$	$[\,]$	x	$\mathrm{pr}_0([\,], \langle y, z\rangle, P_1.P_0)$
$\Longrightarrow^*$	x'	ϵ	$\mathrm{pr}_0([\,], \langle y, z\rangle, P_1.P_0)$
$\Longrightarrow_5$	$[\,]$	$\langle y, z\rangle$	$\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow_4$	$[\,]$	y	$\mathrm{pr}_0([\,], z, \epsilon).\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow^*$	y'	ϵ	$\mathrm{pr}_0([\,], z, \epsilon).\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow_5$	$[\,]$	z	$\mathrm{pr}_1(y', \epsilon).\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow^*$	z'	ϵ	$\mathrm{pr}_1(y', \epsilon).\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow_6$	$\langle y', z'\rangle$	ϵ	$\mathrm{pr}_1(x', P_1.P_0)$
$\Longrightarrow_6$	$\langle x', \langle y', z'\rangle\rangle$	$P_1.P_0$	$[\,]$
$\Longrightarrow_3$	$\langle y', z'\rangle$	P_0	$[\,]$
$\Longrightarrow_2$	y'	ϵ	$[\,]$
$\Longrightarrow_8$		STOP	

The $*$ in the above evaluation represents a series of state transitions executed by the machine before arriving at the given state. The specific state transitions have been omitted to aide in reading and understanding the operation of the machine.

Notice that the basic types $\times$ and 1 are evaluated in an eager fashion. Each component of the pair is evaluated by the machine before the pair is constructed.

4.6. Function parameters

Charity does not have higher order functions, but functions can still be passed as parameters and must be treated with care in the abstract machine. Passing functions amounts to passing the name of a function which will be expanded to combinators when needed at run time. The first step in passing a function involves keeping track of the scope of the parameters. The method employed in this machine is similar to function parameter passing in traditional imperative languages [ASU85]: an activation record is created to hold the state of the machine (also referred to as the context) before a function call. After the function call has been completed, the state of the machine is restored. The machine must save the context so that the calling function will continue as expected after the return form the callee.

The Charity abstract machine uses a special f stack to hold a set of functions passed as parameters to another function. A set of functions passed from a calling function to a callee function is known as a frame. These frames are pushed onto the f stack when a function is called and popped off when the called function completes execution.

To facilitate function passing, two new instructions need to be introduced: *call* and $sel\{\}_i$. The *call* combinator calls a function, pushing the frame of functions onto the f stack. The *sel* combinator selects a function from the current frame on the top of the f stack.

The translation from term logic to combinators follows:

$$\begin{array}{ll} (T_7) & \mathcal{T}\left[v \mapsto f\{v_1 \mapsto t_1, ..., v_n \mapsto t_n\}(t)\right] = \\ & \langle \mathcal{T}\left[v \mapsto t\right], I\rangle; call(f, (\mathcal{T}\left[(v_1, v) \mapsto t_1\right], ..., \mathcal{T}\left[(v_n, v) \mapsto t_n\right])) \\ (T_8) & \mathcal{T}\left[v \mapsto f(t)\right] = \langle \mathcal{T}\left[v \mapsto t\right], \mathcal{T}\left[v \mapsto \sigma\right]\rangle; call(f, ()) \\ (T_9) & \mathcal{T}\left[v \mapsto f_i(t)\right]_{f_1,...,f_n} = \langle \mathcal{T}\left[v \mapsto t\right], \mathcal{T}\left[v \mapsto \sigma\right]\rangle; sel_i \end{array}$$

A call to a function can appear in two different forms: calling with function parameters and without function parameters. Translation T_7 calls the function f, which has function parameters. Notice that each functional parameter has access to the variable base v, as the function may require access to the environment. The extra σ is a special variable used to access the external environment variables, those variables outside the scope of the function passed as a parameter. For example, in the call

```
f (x) = f' {g(x)}().
```

where f' accepts one function parameter, but the variable `x` is outside the scope of `f'`. To access this variable, translation T_7 creates an environment which the code in `f'` can now refer to. Translation T_8 calls the function f, but does not pass any functions as parameters. Translation T_9 selects the appropriate function parameter when it is encountered in the code.

Function definitions also have two forms, with function parameters and without and function parameters. Function parameters passed to a function may require access to the variables of the calling functions as described above. To solve this variable scope problem, the calling function uses the special variable σ as the external environment. For example, a function containing several function parameters:

```
def foo{f, g, h} (v) = t
```

will begin the translation process as:

$$\mathcal{T}\left[(v,\sigma)\mapsto t\right]_{f,g,h}$$

Notice that the translation assumes t will be passed a pair, with the second component σ providing access to the external environment.

The state transitions make use of the f stack to load the functions that are being passed by a call. The *sel* combinator selects the proper function and reloads the scope this function acts in.

	v	c	$d\|f$	$\rightarrow$	v'	c'	$d'\|f'$
10	v	$call\{c',(c_1,...,c_n)\}.c$	d	$\rightarrow$	v	c'	$\mathrm{RET}\{c\}.d$
			f				$\mathrm{arg}\{c_1,...,c_n\}.f$
11	v	$sel\{i\}.c$	d	$\rightarrow$	v	c_i	$\mathrm{reload}(\mathrm{arg}\{c_1,...,c_n\}).\mathrm{cont}(c).d$
			$\mathrm{arg}\{c_1,...,c_n\}.f$				f

Table 4.1. State transition rules for the calling a function

	v	c	$d\|f$	$\rightarrow$	v'	c'	$d'\|f'$
12	v	ϵ	$\text{reload}(\text{arg}\{c_1,...,c_n\}).d$	$\rightarrow$	v	ϵ	d
			f				$\text{arg}\{c_1,...,c_n\}.f$
13	v	ϵ	$\text{ret}(c).d$	$\rightarrow$	v	c	d
			$\text{arg}\{c_1,...,c_n\}.f$				f

TABLE 4.2. State transition rules for the returning from a function

In transition 10, the f stack is loaded with the functions passed to c'. These functions form a *frame*. Essentially, the machine pushes a frame onto the f stack only when passing functions to another function. When one of the functions c_i is encountered in c', the appropriate function from the current frame in the f stack is loaded into the code stack and evaluated. At this point, the machine must load up the previous frame of c_i, so that this function can executed with the proper f stack. When c_i has completed evaluation, the frame for c' must be reloaded into the f stack by transition 12. Finally, when returning from a function call, the current frame on the f stack must be popped off.

To illustrate how passing functions work in this machine, consider the following two Charity functions and their translation to combinator expressions. The function `foo'` simply applies a macro to `x`. The interesting thing about this function is that since `foo'` accepts a macro, it expects that it will be passed the value for `x` along with the environment of the calling function. In `foo'` the environment is discarded with the P_0 instruction and the macro is applied directly to the value of `x`.

```
def foo' {f} (x)   = f(x).
```

$$\begin{aligned}
\text{foo'}: \quad \mathcal{T}\left[(x,\sigma)\mapsto f(x)\right]_f &\Rightarrow \langle \mathcal{T}\left[(x,\sigma)\mapsto x\right], \mathcal{T}\left[(x,\sigma)\mapsto \sigma\right]\rangle; sel_0\\
&\Rightarrow \langle P_0; \mathcal{T}\left[x\mapsto x\right], \mathcal{T}\left[(x,\sigma)\mapsto \sigma\right]\rangle; sel_0\\
&\Rightarrow \langle P_0; I, \mathcal{T}\left[(x,\sigma)\mapsto \sigma\right]\rangle; sel_0\\
&\Rightarrow \langle P_0; I, P_1; \mathcal{T}\left[\sigma\mapsto \sigma\right]\rangle; sel_0\\
&\Rightarrow \langle P_0; I, P_1; I\rangle; sel_0
\end{aligned}$$

The `foo` function passes an abstraction to `foo'`. The abstraction refers to the variable `y`, which is outside its scope. Thus, the environment of `foo` must be passed along with the abstraction.

```
def foo(y) = foo'{x => (x,y)}(y).
```

$$\begin{aligned}
\text{foo}\quad \mathcal{T}\left[y \mapsto foo'\{x \mapsto (x,y)\}(y)\right] &\Rightarrow \langle \mathcal{T}\left[y \mapsto y\right], I\rangle; call(foo', \mathcal{T}\left[(x,y) \mapsto (x,y)\right]) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\mathcal{T}\left[(x,y) \mapsto (x,y)\right])) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\langle \mathcal{T}\left[(x,y) \mapsto x\right], \mathcal{T}\left[(x,y) \mapsto y\right]\rangle)) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\langle P_0; \mathcal{T}\left[x \mapsto x\right], \mathcal{T}\left[(x,y) \mapsto y\right]\rangle)) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\langle P_0; I, \mathcal{T}\left[(x,y) \mapsto y\right]\rangle)) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\langle P_0; I, P_1; \mathcal{T}\left[y \mapsto y\right]\rangle)) \\
&\Rightarrow \langle I, I\rangle; call(foo', (\langle P_0; I, P_1; I\rangle))
\end{aligned}$$

The execution of $foo(8)$ by the abstract machine is shown below:

	v	c	$d \mid f$
	$[\,]$	$8; \langle I, I\rangle; call(foo', (\langle p_0; I, p_1; I\rangle))$	$[\,]$
$\Rightarrow^*$	8	$\langle I, I\rangle; call(foo', (\langle p_0; I, p_1; I\rangle))$	$[\,]$
$\Rightarrow^*$	$\langle 8, 8\rangle$	$call(foo', (\langle p_0; I, p_1; I\rangle))$	$[\,]$

The call to foo' sets up the return from foo' by pushing the empty code stream on the dump (the code after the call combinator) and pushing the function arguments onto the f stack.

	v	c	$d \mid f$
$\Rightarrow$	$\langle 8, 8\rangle$	$\langle P_0; I, P_1; I\rangle; sel_0$	$ret(\epsilon)$ $arg(\langle p_0; I, p_1; I\rangle$
$\Rightarrow$	$\langle 8, 8\rangle$	sel_0	$[\,]$ $arg(\langle p_0; I, p_1; I\rangle$

A sel_0 instruction will load the code from the f stack onto the code stack.

	v	c	$d \mid f$
$\Rightarrow$	$\langle 8, 8\rangle$	$\langle P_0; I, P_0; I\rangle$	$arg(\langle p_0; I, p_1; I\rangle).ret(\epsilon)$
$\Rightarrow$	$\langle 8, 8\rangle$	ϵ	$arg(\langle p_0; I, p_1; I\rangle).ret(\epsilon)$
$\Rightarrow$	$\langle 8, 8\rangle$	ϵ	$ret(\epsilon)$
			$arg(\langle p_0; I, p_1; I\rangle)$
$\Rightarrow$	$\langle 8, 8\rangle$	ϵ	[ ]
$\Rightarrow$			STOP

As expected, the computation results in $(8, 8)$ being left on top the of v stack.

Chapter 5

The Charity Abstract Machine : datatypes

The Charity system can be customized by adding datatypes. These are divided into the inductive datatypes and their dual, the coinductive datatypes. Immediately following a datatype declaration, three combinators (map, fold/unfold, case/record) for manipulating the datatypes are delivered to the system. Similarly to the basic combinators, those associated with the datatypes access variables in an environment similar to the basic combinators. This chapter shows how the inductive and coinductive datatypes and the combinators that manipulate their elements are added to the basic Charity system.

5.1. Inductive datatypes

When an inductive datatype L is declared, constructors for building elements of the datatype, along with three combinators (for the case, fold, and map), are immediately delivered to the system. Recall the definition of an inductive datatype is of the form:

$$\begin{array}{rlcl} \text{data } L(A) \longrightarrow S = & c_1 : E_1(A,S) & \longrightarrow & S \\ | & \vdots & & \\ | & c_n : E_n(A,S) & \longrightarrow & S. \end{array}$$

In each constructor c_i of L, the domain type expression $E_i(A,S)$ is a type where

- A is the parametric variable,
- S is the state variable,

such that

$$\begin{array}{rcll} E(A,S) & ::= & A & \text{parametric variable} \\ & | & S & \text{state variable} \\ & | & 1 & \text{unit (basic type)} \\ & | & E(A,S) \times E(A,S) & \text{product (basic type)} \\ & | & L(E(A,S),...,E(A,S)) & \text{other datatype (user defined type)} \end{array}$$

5.1.1. Strong datatypes

Adding strength to datatypes provides a means of accessing global values outside the scope of a data structure. In this scheme, the environment is distributed uniformly over the structure of the datatype. This means distributing the environment over the constructors of a data structure. However, constructors are simply combinators with types:

$$c_i : E_i(A, L(A)) \rightarrow L(A)$$

For example, the cons constructor from the list datatype is:

$$cons : A \times list(A) \rightarrow list(A)$$

Hence, distributing the environment into the local context of a constructor is viewed as distributing the environment over a combinator.

To "strengthen" a combinator by distributing the environment into its local context, three maps (also combinators) are provided. These maps distribute the environment into a single level of the combinator and are not recursive in nature. The parametric component of a constructor's type expression $E_i(A, S)$ is strengthened with the map[1]:

$$\theta^{E_i}_{\neg R} : E_i(A, S) \times \sigma \rightarrow E_i(A \times \sigma, S)$$

[1] The subscripted $\neg R$ refers to the nonrecursive parametric component.

while the state component is strengthened with[2]:

$$\theta_R^{E_i} : E_i(A,S) \times \sigma \to E_i(A, S \times \sigma)$$

or in case both components are strengthened, we use:

$$\theta^{E_i} : E_i(A,S) \times \sigma \to E_i(A \times \sigma, S \times \sigma)$$

As in the last chapter, the environment is represented by σ. In short, adding strength to a component of a type expression will pair up each occurrence of the component in the current level of a data structure with a copy of the environment. For example, suppose *list* is a type of arity 1, and A, B are parametric variables, and S the state variable in the type expression

$$E(A,S) \quad = \quad A \times list(B \times S)$$

Then applying the above three maps would have the following actions:

$$\begin{array}{rlcl} \theta_{\neg R}^{E_i}(E): & (A \times list(B \times S)) \times \sigma & \to & (A \times \sigma) \times list((B \times \sigma) \times S) \\ \theta_R^{E_i}(E): & (A \times list(B \times S)) \times \sigma & \to & A \times list(B \times (S \times \sigma)) \\ \theta^{E_i}(E): & (A \times list(B \times S)) \times \sigma & \to & (A \times \sigma) \times list((B \times \sigma) \times (S \times \sigma)) \end{array}$$

Notice that when a datatype is encountered in the type expression (such as the *list* in the above example), the parameters passed into the datatype are also paired with the environment in the same way.

To apply a function to the parametric component or the state component of a term, the following combinator is used:

$$E_i\{f, g\} : E_i(A,S) \to E_i(A', S')$$

where $f : A \to A'$ and $g : S \to S'$. The $E_i\{\}$ combinator should not be confused with the type $E_i()$. Combinators acting over a data structure require an environment.

[2] The subscripted R refers to the recursive state component.

Therefore, the system is supplied with another combinator:

$$map^{E_i}\{f, g\} = \theta^{E_i}; E_i\{f, g\}$$

For example, given the constructor $c_j : E_j(A, S)$, $f : A \to A'$, and $g : S \to S'$, the map^{E_j} combinator will be:

$$map^{E_j}\{f, g\} : E_j(A, S) \times \sigma \to E_j(A', S')$$

where each component of the term E_j is strengthened, then transformed.

5.1.2. Mapping over inductive datatypes

The map combinator is important to Charity programming. Given an element of a datatype such as the list $[a_0, ..., a_n]$, the map operation on the list would apply a function f to every element, eg:

$$map\ \{f\}\ [a_0, ..., a_n] = [f(a_0), ..., f(a_n)]$$

In general, applying a map over an inductive datatype in Charity requires that the term for each constructor of the datatype be "mapped" with the map^{E_i} combinator. Given a datatype L, a map combinator for L has type:

$$map^L\{A_1 \times \sigma \to A'_1, ..., A_m \times \sigma \to A'_m\} \quad : \quad L(A_1, ..., A_m) \times \sigma \to L(A'_1, ..., A'_m)$$

where m is the number of parametric types of L. The map combinator is then:

$$map^L\{g1, ..., g_m\} = \theta^L; L\{g_1, ..., g_m\}$$

where θ^L recursively strengthens the entire data structure of L, and then $L\{g_1, ..., g_n\}$ applies a map function over L as described earlier in this section.

The commuting diagram for the map combinator below shows how the environment is distributed over the entire datatype:

$$\begin{tikzcd}
E_i(A, L(A)) \times \sigma \arrow[rr, "c_i \times I"] \arrow[d, "\langle \theta^{E_i}_{R}, P_1 \rangle"'] & & L(A) \times \sigma \arrow[dd, dashed, "\theta^L"] \\
E_i(A, L(A) \times \sigma) \times \sigma \arrow[d, dashed, "E_i\{I, \theta^L\} \times I"'] & & \\
E_i(A, L(A \times \sigma)) \times \sigma \arrow[r, "\theta^{E_i}_{\neg R}"'] & E_i(A \times \sigma, L(A \times \sigma)) \arrow[r, "c_i"'] & L(A \times \sigma)
\end{tikzcd}$$

Note that $a \times b$ is shorthand for $\langle P_0; a, P_1; b\rangle$. Hence,

$$\theta^{E_i}\{I, \theta^L\} \times I = \langle P_0; \theta^{E_i}\{I, \theta^L\}, P_1; I\rangle$$

The above diagram shows how the environment is distributed into the entire data structure in two ways. Following the top and left path provides the abstract, general view of distributing the environment with the combinator θ^L. The abstract view can be rewritten in a step wise manner as follows:

(1) strengthen the state component and propagate the environment,

(2) recursively apply the environment distribution combinator θ^L to the recursive component and propagate the environment,

(3) strengthen the parametric component,

(4) apply the constructor combinator c_i to build a strengthened data structure $L(A \times \sigma)$.

The rewrite rule for the map combinator distributes the environment before applying the mapped function. This rewrite rule is derived in [CF92] and shown below:

$$\begin{aligned}
c_i \times I; \mathrm{map}^L\{g_1, ..., g_n\} &\Longrightarrow c_i \times I; \theta^L; L\{f\} \\
&\Longrightarrow \langle \theta_R^{E_i}, P_1 \rangle; E_i\{I, \theta^L\} \times I; \theta_{\neg R}^{E_i}; c_i; L\{f\} \\
&\Longrightarrow \langle \theta_R^{E_i}, P_1 \rangle; \theta_{\neg R}^{E_i}; E_i\{I, \theta^L\}; c_i; L\{f\} \\
&\Longrightarrow \theta^{E_i}; E_i\{I, \theta^L\}; c_i; L\{f\} \\
&\Longrightarrow \theta^{E_i}; E_i\{I, \theta^L\}; E_i\{f, L\{f\}\}; c_i \\
&\Longrightarrow \theta^{E_i}; E_i\{f; \theta^L; L\{f\}\}; c_i \\
&\Longrightarrow map^{E_i}\{f, map^L\{f\}\}; c_i
\end{aligned}$$

This rewrite rule can be interpreted as "at the current level in a data structure tree, apply f to the parametric components of the structure and recursively apply the map^L combinator to every state component of the structure".

Mapping over types 1 and product

To map over the basic types 1 and products, two new combinators are introduced into the system:

$$\begin{aligned}
map^1\{\} &: X \times \sigma \to X \\
map^\times\{X \times \sigma \to X', Y \times \sigma \to Y'\} &: (X \times Y) \times \sigma \to X' \times Y'
\end{aligned}$$

Since the type 1 does not have any input parameters, the map over 1 simply eliminates the strength. The map over the product type strengthens each component of the pair (by distributing the environment) before applying the mapped functions to the two parametric values in the pair combinator.

5.1.3. The fold combinator

The purpose of the fold combinator is to provide a mechanism for transforming an element of one user defined datatype to an element of another user defined datatype. The fold combinator has type:

$$fold^L\{E_1(A, C) \times \sigma \to C, ..., E_n(A, C) \times \sigma \to C\} \;:\; L(A) \times \sigma \to C$$

where the input combinators need to be strengthened with the environment σ in the same way as the map combinator.

The following commuting diagram derived in [Spe93] defines the action of the fold combinator:

$$\begin{tikzcd}
E_i(A, L(A)) \times \sigma \arrow[r, "c_i \times I"] \arrow[d, "\langle \theta_R^{E_i}, P_1 \rangle"'] & L(A) \times \sigma \arrow[dd, dashed, "\mathrm{fold}^L"] \\
E_i(A, L(A) \times \sigma) \times \sigma \arrow[d, dashed, "E_i\{I, \mathrm{fold}^L\} \times I"'] & \\
E_i(A, Y) \times \sigma \arrow[r, "g_i"] & Y
\end{tikzcd}$$

where each $g_i : E_i(A, Y) \times \sigma \rightarrow Y$ is a user defined phrase used for transforming elements of type L to elements of y.

For the above fold diagram, following the path of the diagram starting from the top left via the upper right side, this can be rewritten as the bottom left path. This rewriting gives the defining rule of the fold combinator. The above diagram can be thought of as:

> Applying the constructor c_i to the type E_i will build an element of the datatype L. Applying fold to L will transform the element of L to an element of Y. However, this can be regarded as first distributing the environment into the recursive (state) component of E_i by applying θ, then applying the fold combinator to each recursive component of E_i. Finally, the phrase g_i will transform the expression into a value of type Y.

The rewrite rule for the fold combinator can be derived from the diagram as follows:

$$\begin{aligned}
& c_i \times I; \mathrm{fold}^L\{g_1, ..., g_n\} \\
\Longrightarrow\quad & \langle\theta_R^{E_i}, P_1\rangle; \underline{E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\} \times I}; g_i \\
\Longrightarrow\quad & \underline{\langle\theta_R^{E_i}, P_1\rangle; \langle P_0; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}, P_1; I\rangle}; g_i \\
\Longrightarrow\quad & \langle\underline{\langle\theta_R^{E_i}, P_1\rangle; P_0}; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}, \langle\theta_R^{E_i}, P_1\rangle; P_1; I\rangle; g_i \\
\Longrightarrow\quad & \langle\theta_R^{E_i}; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}, \underline{\langle\theta_R^{E_i}, P_1\rangle; P_1}; I\rangle; g_i \\
\Longrightarrow\quad & \langle\theta_R^{E_i}; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}, \underline{P_1; I}\rangle; g_i \\
\Longrightarrow\quad & \langle\underline{\theta_R^{E_i}}; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}, P_1\rangle; g_i \\
\Longrightarrow\quad & \langle\theta^{E_i}; \underline{E_i\{P_0, I\}; E_i\{I, \mathrm{fold}^L\{g_1, ..., g_n\}\}}, P_1\rangle; g_i \\
\Longrightarrow\quad & \langle\theta^{E_i}; E_i\{\underline{P_0; I}, I; \mathrm{fold}^L\{g_1, ..., g_n\}\}, P_1\rangle; g_i \\
\Longrightarrow\quad & \langle\theta^{E_i}; E_i\{P_0, \underline{I; \mathrm{fold}^L\{g_1, ..., g_n\}\}}, P_1\rangle; g_i \\
\Longrightarrow\quad & \langle\underline{\theta^{E_i}; E_i\{P_0, \mathrm{fold}^L\{g_1, ..., g_n\}\}}, P_1\rangle; g_i \\
\Longrightarrow\quad & \langle map^{E_i}\{P_0, \mathrm{fold}^L\{g_1, ..., g_n\}\}, P_1\rangle; g_i
\end{aligned}$$

The fold combinator is a tail recursive function applied to all state components of a term before the phrase g_i is applied.

5.1.4. The case combinator

The case operation is a special instance of the fold, where the maps are only applied to the "head" value of the data structure (ie. it is non–recursive). The combinator for the case combinator is:

$$case^L\{E_1(A, L(A)) \times \sigma \to C, ..., E_n(A, L(A)) \times \sigma \to C\} \quad : \quad L(A) \times \sigma \to C$$

The rewrite rule for the case combinator is:

$$c_i \times \sigma; case\{h_1, ..., h_n\} \to h_i$$

where $h_i : E_i(A, L(A)) \times \sigma \to C$.

5.2. Translation to combinators

After the translation from term logic to combinators extended below, all free variables are eliminated from the resultant combinator expression. The environment passing mechanism for the inductive combinators is the same as that for abstractions. Before the inductive combinator is called, a pair is created, where the first component is the input, while the second component is the environment.

$(\mathcal{T}_{10})\quad \mathcal{T}\left[v \mapsto c_i(t)\right] = \mathcal{T}\left[v \mapsto t\right]; c_i$ where c is a constructor,

$$(\mathcal{T}_{11})\quad \mathcal{T}\left[v \mapsto \left\{\begin{array}{ccc} c_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ c_n(v_n) & \mapsto & t_n \end{array}\right\}(t)\right] = \begin{array}{ll} \langle \mathcal{T}\left[v \mapsto t\right], I\rangle; case^L\{ & \mathcal{T}\left[(v_1, v) \mapsto t_1\right], ..., \\ & \mathcal{T}\left[(v_n, v) \mapsto t_n\right] \}, \end{array}$$

$$(\mathcal{T}_{12})\quad \mathcal{T}\left[v \mapsto \left\{\!\!\left\{\begin{array}{ccc} c_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ c_n : v_n & \mapsto & t_n \end{array}\right\}\!\!\right\}(t)\right] = \begin{array}{ll} \langle \mathcal{T}\left[v \mapsto t\right], I\rangle; fold^L\{ & \mathcal{T}\left[(v_1, v) \mapsto t_1\right], ..., \\ & \mathcal{T}\left[(v_n, v) \mapsto t_n\right] \}, \end{array}$$

$$(\mathcal{T}_{13})\quad \mathcal{T}\left[v \mapsto L\left\{\begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array}\right\}(t)\right] = \begin{array}{ll} \langle \mathcal{T}\left[v \mapsto t\right], I\rangle; map^L\{ & \mathcal{T}\left[(v_1, v) \mapsto t_1\right], ..., \\ & \mathcal{T}\left[(v_m, v) \mapsto t_m\right] \}, \end{array}$$

where n is the number of constructions and m is the number of parametric variables.

5.3. Rewrite rules and machine transitions

The rewrite rules for the basic types as described in section 5.1.2 are as follows:

$$\begin{array}{lrcl} (R_7) & \langle v_0, v_1\rangle; map^1\{\} & \Longrightarrow & v_0 \\ (R_8) & \langle\langle v_0, v_1\rangle, \sigma\rangle; map^{\times}\{f, g\} & \Longrightarrow & \langle\langle v_0, \sigma\rangle, \langle v_1, \sigma\rangle\rangle; f \times g \end{array}$$

These rules produce the corresponding state transitions for the abstract machine:

	v	c	$d \mid f$		v'	c'	$d' \mid f'$
14	$\langle v, \sigma\rangle$	$map^1\{\}.c$	d	$\longrightarrow$	v	c	d
15	$\langle\langle v_0, v_1\rangle, \sigma\rangle$	$map^{\times}\{f, g\}.c$	d	$\longrightarrow$	$\langle v_0, \sigma\rangle$	f	$\mathrm{pr}_0(\langle v_1, \sigma\rangle, g, c).d$

Notice that the state transition for the $map^{\times}$ combinator maps f over the first component and saves g on the dump so that it can be mapped over the second component of the pair. In addition, each component of the pair is strengthened with the environment σ.

Listed below is a summary of the rewrite rules associated with inductive datatypes (case, fold and map).

$$\begin{array}{lll}
(R_9) & \langle c_i; v, \sigma\rangle; case^L\{f_1, ..., f_n\} & \Longrightarrow \langle v, \sigma\rangle; f_i \\
(R_{10}) & \langle c_i; v, \sigma\rangle; fold^L\{g_1, ..., g_n\} & \Longrightarrow \langle v, \sigma\rangle; \langle map^{E_i}\{P_0, fold^L\{g_1, ..., g_n\}\}, P_1\rangle; g_i \\
(R_{11}) & \langle c_i; v, \sigma\rangle; map^L\{h_1, ..., h_m\} & \Longrightarrow \langle v, \sigma\rangle; map^{E_i}\{h_1, ..., h_m, \\
 & & \qquad map^L\{h_1, ..., h_m\}\}; c_i
\end{array}$$

For the above rules, the pair on the value stack holds the data structure in the first component, and environment in the second component.

The corresponding state transitions derived directly from the above rewrite rules are:

	v	c	$d \vert f$		v'	c'	$d' \vert f'$
16	$\langle c_i.v, \sigma\rangle$	$case^L\{f_1, ..., f_n\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	f_i	cont$(c).d$
17	$\langle c_i.v, \sigma\rangle$	$fold^L\{g_1, ..., g_n\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$\langle map^{E_i}\{P_0, fold^L\{g_1, ..., g_n\}\},$ $P_1\rangle.g_i$	cont$(c).d$
18	$\langle c_i.v, \sigma\rangle$	$map^L\{h_1, ..., h_m\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$map^{E_i}\{h_1, ..., h_m,$ $map^L\{h_1, ..., h_m\}\}.c_i$	cont$(c).d$
19	v	$u.c$	d	$\longrightarrow$	$u.v$	c	d

An additional state transition, 19, stacks constructors onto the value stack. This transition also applies to coinductive combinators described in section 5.5.

5.4. Examples of inductive datatypes

This section provides a number of examples of how defining inductive datatypes will deliver the case, fold and map combinators to the system. In addition, they illustrate the translation from term logic to combinators for each of the inductive operations.

5.4.1. Booleans

Recall the definition of the boolean datatype:

```
data bool -> C = true : 1 -> C
               | false: 1 -> C.
```

which says that the inductive datatype `bool` with no parametric variables and a state variable C has constructors with domain types:

$$true : E_{true}(\_, C) = 1$$

$$false : E_{false}(\_, C) = 1$$

$$\begin{tikzcd}
1 \times \sigma \arrow[r, "true \times I"] \arrow[dr, "f_T"'] & bool \times I \arrow[d, dashed, "h"'] & 1 \times \sigma \arrow[l, "false \times I"'] \arrow[dl, "f_F"] \\
 & Y &
\end{tikzcd}$$

where $h = fold\{f_T, f_F\}$.

Since the definition of `bool` does not contain state variables, the fold diagram is not recursive. Hence, the case combinator and the fold combinator perform the same action. In addition, the fact that there is no parametric variables means the the map combinator acts in the same manner as the identity function.

Consider a simple example using the `and` function:

```
def and(x,y) =
   { true () => y
   | false() => false
   } (x).
```

This will result in the following translation:

$$\underline{\mathcal{T}\left[(x,y)\mapsto\left\{\begin{array}{ccc} true() & \mapsto & y \\ false() & \mapsto & false() \end{array}\right\}(x)\right]}$$

$$\begin{array}{ll}
\Longrightarrow_{T_8} & \langle \underline{\mathcal{T}\left[(x,y)\mapsto x\right]}, I\rangle; case^{bool}\{\mathcal{T}\left[((),(x,y))\mapsto y\right], \mathcal{T}\left[((),(x,y))\mapsto false()\right]\} \\
\Longrightarrow_{T_3} & \langle P_0; \underline{\mathcal{T}\left[x\mapsto x\right]}, I\rangle; case^{bool}\{\mathcal{T}\left[((),(x,y))\mapsto y\right], \mathcal{T}\left[((),(x,y))\mapsto false()\right]\} \\
\Longrightarrow_{T_2} & \langle P_0; I, I\rangle; case^{bool}\{\underline{\mathcal{T}\left[((),(x,y))\mapsto y\right]}, \mathcal{T}\left[((),(x,y))\mapsto false()\right]\} \\
\Longrightarrow_{T_3} & \langle P_0; I, I\rangle; case^{bool}\{P_1; \underline{\mathcal{T}\left[(x,y)\mapsto y\right]}, \mathcal{T}\left[((),(x,y))\mapsto false()\right]\} \\
\Longrightarrow_{T_3} & \langle P_0; I, I\rangle; case^{bool}\{P_1; P_1; \underline{\mathcal{T}\left[y\mapsto y\right]}, \mathcal{T}\left[((),(x,y))\mapsto false()\right]\} \\
\Longrightarrow_{T_2} & \langle P_0; I, I\rangle; case^{bool}\{P_1; P_1; I, \underline{\mathcal{T}\left[((),(x,y))\mapsto false()\right]}\} \\
\Longrightarrow_{T_6} & \langle P_0; I, I\rangle; case^{bool}\{P_1; P_1; I, \underline{\mathcal{T}\left[((),(x,y))\mapsto ()\right]}; false\} \\
\Longrightarrow_{T_1} & \langle P_0; I, I\rangle; case^{bool}\{P_1; P_1; I, !; false\}
\end{array}$$

The rewrite rules for `bool` are:

$$\begin{array}{l}
\langle c_i; v, \sigma\rangle; case^{bool}\{f_T, f_F\} \Longrightarrow \langle v, \sigma\rangle; f_i \\
\langle c_i; v, \sigma\rangle; fold^{bool}\{f_T, f_F\} \Longrightarrow \langle v, \sigma\rangle; \langle map^1\{\}, P_1\rangle; f_i \\
\langle c_i; v, \sigma\rangle; map^{bool}\{\} \Longrightarrow \langle v, \sigma\rangle; map^1\{\}; c_i
\end{array}$$

where $i \in bool$. Hence the state transitions produced by these rules are:

v	c	$d\|f$		v'	c'	$d'\|f'$
$\langle c_i.v, \sigma\rangle$	$case^{bool}\{f_T, f_F\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	f_{c_i}	cont$(c).d$
$\langle c_i; v, \sigma\rangle$	$fold^{bool}\{f_T, f_F\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$\langle map^1\{\}, P_1\rangle; f_i$	cont$(c).d$
$\langle c_i; v, \sigma\rangle$	$map^{bool}\{\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$map^1\{\}.c_i$	cont$(c).d$

When the abstract machine evaluates the $case^{bool}$ combinator, the appropriate phrase is selected and evaluated, depending on the input constructor. The case over `bools` provides the if–then–else construct for providing conditional control.

5.4.2. Natural numbers

The natural numbers are defined as:

```
data nat() -> C = zero: 1 -> C
                | succ: C -> C.
```

similarly to the bool datatype, there are no parametric variables. However, the domain type of the the succ constructor contains a state variable, yielding a recursive definition. The fold diagram shows how recursion is handled:

$$\begin{tikzcd}
1 \times \sigma \arrow[r, "zero \times I"] \arrow[dr, "f"'] & nat \times \sigma \arrow[d, dashed, "fold^{nat}\{f,g\}"] & nat \times \sigma \arrow[l, "succ \times I"'] \arrow[d, dashed, "\langle fold^{nat}\{f,g\}\,, P_1\rangle"] \\
& Y & Y \times \sigma \arrow[l, "g"]
\end{tikzcd}$$

The rewrite rules for the fold over the natural numbers are:

$$\begin{array}{lll}
\text{zero} & & \langle zero; v, I\rangle; fold^{nat}\{f,g\} \\
& \Longrightarrow & \langle zero; v, I\rangle; \langle map^{E_{zero}}\{P_0, fold^{nat}\{f,g\}, P_1\rangle; f \\
& \Longrightarrow & \langle zero; v, I\rangle; \langle map^{1}\{\}, P_1\rangle; f
\end{array}$$

$$\begin{array}{lll}
\text{succ} & & \langle succ; v, I\rangle; fold^{nat}\{f,g\} \\
& \Longrightarrow & \langle succ; v, I\rangle; \langle map^{E_{succ}}\{P_0, fold^{nat}\{f,g\}, P_1\rangle; g \\
& \Longrightarrow & \langle succ; v, I\rangle; \langle fold^{nat}\{f,g\}, P_1\rangle; g
\end{array}$$

where f and g are user defined phrases. In the succ case, tail recursion is performed until the zero constructor is encountered and the zero phrase is executed. Then the change propagates upwards, executing the succ phrase at each stage of the return.

The state transitions generated for the abstract machine are:

v	c	d		v'	c'	d'
$\langle \mathrm{zero}.v, \sigma\rangle$	$fold^{nat}\{f,g\}.c$	d	$\rightarrow$	$\langle v, \sigma\rangle$	$\langle map^{1}\{\}, P_1\rangle.f$	$\mathrm{cont}(c).d$
$\langle \mathrm{succ}.v, \sigma\rangle$	$fold^{nat}\{f,g\}.c$	d	$\rightarrow$	$\langle v, \sigma\rangle$	$\langle fold^{nat}\{f,g\}, P_1\rangle.g$	$\mathrm{cont}(c).d$

The function which adds two numbers together is defined as:

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

The translation of `add` follows:

$$\underline{\mathcal{T}\left[(x,y) \mapsto \left\{\!\!\left|\begin{array}{lcl} zero : () & \mapsto & y \\ succ : n & \mapsto & succ(n) \end{array}\right|\!\!\right\}(x)\right]}$$

$$\Longrightarrow_{T_9} \langle \underline{\mathcal{T}\,[(x,y) \mapsto x]}, I\rangle; fold^{nat}\{\mathcal{T}\,[((),(x,y)) \mapsto y]\,, \mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]\}$$

$$\Longrightarrow_{T_3} \langle P_0; \underline{\mathcal{T}\,[x \mapsto x]}, I\rangle; fold^{nat}\{\mathcal{T}\,[((),(x,y)) \mapsto y]\,, \mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]\}$$

$$\Longrightarrow_{T_2} \langle P_0; I, I\rangle; fold^{nat}\{\underline{\mathcal{T}\,[((),(x,y)) \mapsto y]}, \mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]\}$$

$$\Longrightarrow_{T_3} \langle P_0; I, I\rangle; fold^{nat}\{P_1; \underline{\mathcal{T}\,[(x,y) \mapsto y]}, \mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]\}$$

$$\Longrightarrow_{T_3} \langle P_0; I, I\rangle; fold^{nat}\{P_1; P_1; \underline{\mathcal{T}\,[y \mapsto y]}, \mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]\}$$

$$\Longrightarrow_{T_2} \langle P_0; I, I\rangle; fold^{nat}\{P_1; P_1; I, \underline{\mathcal{T}\,[(n,(x,y)) \mapsto succ(n)]}\}$$

$$\Longrightarrow_{T_6} \langle P_0; I, I\rangle; fold^{nat}\{P_1; P_1; I, \underline{\mathcal{T}\,[(n,(x,y)) \mapsto n]}; succ\}$$

$$\Longrightarrow_{T_3} \langle P_0; I, I\rangle; fold^{nat}\{P_1; P_1; I, P_0; \underline{\mathcal{T}\,[n \mapsto n]}; succ\}$$

$$\Longrightarrow_{T_3} \langle P_0; I, I\rangle; fold^{nat}\{P_1; P_1; I, P_0; I; succ\}$$

5.4.3. Lists

Lists are defined in Charity as:

```
data list(A) -> C = nil : 1 -> C
                  | cons: A * C -> C.
```

lists provide a more complex datatype example where both type parameters and

recursive definitions are present. The fold diagram for lists is:

$$\begin{tikzcd}
1 \times \sigma \arrow[r, "nil \times I"] \arrow[dr, "f"'] & list(A) \times \sigma \arrow[d, "fold^{list}\{f,g\}"] & (A \times list(A)) \times \sigma \arrow[l, "cons \times I"'] \arrow[d, "\langle \theta_R^{\times}; 1 \times fold^{list}\{f,g\}, P_1 \rangle"] \\
 & Y & (A \times Y) \times \sigma \arrow[l, "g"]
\end{tikzcd}$$

The maps for each constructor are:

$$\begin{aligned}
map^E_{nil}(f_A, f_C) &= map^1\{\} \\
map^E_{cons}(f_A, f_C) &= map^{\times}\{f_A, f_C\}
\end{aligned}$$

The rewrite rules generated for the case, fold, and map are:

case

$$\langle c_i; v, \sigma\rangle; case^{list}\{f_0, f_1\} \quad \Longrightarrow \quad \langle v, \sigma\rangle; f_i$$

fold

$$\begin{aligned}
\langle \mathrm{nil}; v, \sigma\rangle; fold^{list}\{g_{nil}, g_{cons}\} \quad &\Longrightarrow \quad \langle v, \sigma\rangle; \langle map^{E_i}\{P_0, fold^{list}\{g_{nil}, g_{cons}\}\}, P_1\rangle; g_{nil} \\
&\Longrightarrow \quad \langle v, \sigma\rangle; \langle P_0, P_1\rangle; g_{nil}
\end{aligned}$$

$$\begin{aligned}
\langle \mathrm{cons}; v, \sigma\rangle; fold^{list}\{g_{nil}, g_{cons}\} \quad &\Longrightarrow \quad \langle v, \sigma\rangle; \langle map^{E_i}\{P_0, fold^{list}\{g_{nil}, g_{cons}\}\}, P_1\rangle; g_{cons} \\
&\Longrightarrow \quad \langle v, \sigma\rangle; \langle map^{\times}\{P_0, fold^{list}\{g_{nil}, g_{cons}\}\}, P_1\rangle; g_{cons}
\end{aligned}$$

map

$$\begin{aligned}
\langle nil; v, \sigma\rangle; map^{list}\{f\} \quad &\Longrightarrow \quad \langle v, \sigma\rangle; map^{E_{nil}}\{f, map^{list}\{f\}; \mathrm{nil} \\
&\Longrightarrow \quad \langle v, \sigma\rangle; map^1\{\}; \mathrm{nil}
\end{aligned}$$

$$\begin{aligned}
\langle cons; v, \sigma\rangle; map^{list}\{f\} \quad &\Longrightarrow \quad \langle v, \sigma\rangle; map^{E_{cons}}\{f, map^{list}\{f\}; \mathrm{cons} \\
&\Longrightarrow \quad \langle v, \sigma\rangle; map^{\times}\{f, map^{list}\{f\}\}; \mathrm{cons}
\end{aligned}$$

These rewrite rules deliver the following state transition rules to the machine:

v	c	$d \vert f$		v'	c'	$d' \vert f'$
$\langle c_i.v, \sigma\rangle$	$case^{list}\{f_0, f_1\}.c$	d	$\longrightarrow$	v	f_i	$\mathrm{cont}(c).d$
$\langle c_i.v, \sigma\rangle$	$fold^{list}\{g_0, g_1\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$\langle map^{E_i}\{P_0, fold^{list}\{g_0, g_1\}\}, P_1\rangle.g_i$	$contc.d$
$\langle c_i.v, \sigma\rangle$	$map^{list}\{f\}.c$	d	$\longrightarrow$	$\langle v, \sigma\rangle$	$\langle map^{E_i}\{P_0, map^{list}\{g_0, g_1\}\}, P_1\rangle.c_i$	$contc.d$

An example of a case combinator determining if a list is empty follows:

```
def isEmpty(L) =
   { nil ()       => true
   | cons(x, l') => false
   } (L).
```

This program is translated to the combinator expression: $\langle I, I\rangle; \mathrm{case}^{list}\{!; \mathrm{true}, !; \mathrm{false}\}$.

The expression `isEmpty([zero])` is evaluated as follows:

	v	c	d
	$[\,]$	$\langle \mathrm{zero}, \mathrm{nil}\rangle; \mathrm{cons}; \langle I, I\rangle; case^{list}\{!; \mathrm{true}, !; \mathrm{false}\}$	$[\,]$
$\Longrightarrow$	$[\,]$	zero	$\mathrm{pr}_0([\,], \mathrm{nil}, \mathrm{cons}; \langle I, I\rangle;$ $case^{list}\{!; \mathrm{true}, !; \mathrm{false}\})$
$\Longrightarrow$	zero	ϵ	$\mathrm{pr}_0([\,], \mathrm{nil}, \mathrm{cons}; \langle I, I\rangle;$ $case^{list}\{!; \mathrm{true}, !; \mathrm{false}\})$

The first component of the pair has been processed and now the machine must pop the code for the second component off the dump and evaluate it:

	v	c	d
$\Longrightarrow$	$[\,]$	nil	$\mathrm{pr}_1(\mathrm{zero}, \mathrm{cons}; \langle I, I\rangle; case^{list}\{!; \mathrm{true}, !; \mathrm{false}\})$
$\Longrightarrow$	nil	ϵ	$\mathrm{pr}_1(\mathrm{zero}, \mathrm{cons}; \langle I, I\rangle; case^{list}\{!; \mathrm{true}, !; \mathrm{false}\})$

Components of the pair have now been evaluated, and the resultant pair must be created on the value stack:

	v	c	d
$\Longrightarrow$	$\langle \mathrm{zero}, \mathrm{nil}\rangle$	$\mathrm{cons}; \langle I, I\rangle; case^{list}\{!; \mathrm{true}, !; \mathrm{false}\}$	$[\,]$
$\Longrightarrow$	$\langle \mathrm{zero}, \mathrm{nil}\rangle.\mathrm{cons}$	$\langle I, I\rangle; case^{list}\{!; \mathrm{true}, !; \mathrm{false}\}$	$[\,]$

The pair on the value stack is now distributed over the pair combinator in the code stream:

	v	c	d
$\Longrightarrow$	$\langle\text{zero}, \text{nil}\rangle.\text{cons}$	ϵ	$\text{pr}_0(\langle\text{zero}, \text{nil}\rangle.\text{cons}, [\,], case^{list}\{!; \text{true}, !; \text{false}\})$
$\Longrightarrow$	$\langle\text{zero}, \text{nil}\rangle.\text{cons}$	ϵ	$\text{pr}_1(\langle\text{zero}, \text{nil}\rangle.\text{cons}, case^{list}\{!; \text{true}, !; \text{false}\})$

Finally, the evaluation of the case combinator strips off the nil constructor and executes the nil phrase of the case. The result of the evaluation is !.false, as expected:

	v	c	d
$\Longrightarrow$	$\langle\langle\text{zero}, \text{nil}\rangle.\text{cons}, \langle\text{zero}, \text{nil}\rangle.\text{cons}\rangle$	$case^{list}\{!; \text{true}, !; \text{false}\}$	$[\,]$
$\Longrightarrow$	$\langle\langle\epsilon, \text{nil}\rangle.\text{cons}, \langle\text{zero}, \text{nil}\rangle.\text{cons}\rangle$	$!; \text{false}$	$ret(\epsilon)$
$\Longrightarrow$	$!$	false	$ret(\epsilon)$
$\Longrightarrow$	$!.\text{false}$	ϵ	$ret(\epsilon)$
$\Longrightarrow$	$!.\text{false}$	ϵ	$[\,]$
$\Longrightarrow$		STOP	

For the next example, consider appending one list to the end of another as is done with the following Charity function:

```
def append(L1, L2) =
   {| nil : ()     => L2
    | cons: (x, L) => cons(x, L)
    |} (L1).
```

The append function is translated, via the translation rules, to:

$$\langle P_0, I\rangle; fold^{list}\{P_1; P_1, \langle P_0; P_0, P_0; P_1\rangle; cons\}$$

Notice that append folds over `L1`. The pair in front of the fold combinator binds it together with the environment. We would expect a product of two lists to be passed to the append function, but the first list is the folded argument. The second list, however, is present in the environment. When the first list is empty, the rewritings for the `nil` phrase are initiated:

$$\begin{aligned}\langle v,\sigma\rangle;\langle P_0,P_1\rangle;g_{nil} &\Longrightarrow \langle v,\sigma\rangle;\langle P_0,P_1\rangle;P_1;P_1\\ &\Longrightarrow \langle v,\sigma\rangle;I;P_1;P_1\\ &\Longrightarrow \langle v,\sigma\rangle;P_1;P_1\\ &\Longrightarrow \sigma;P_1\end{aligned}$$

This means that given the original two lists passed into the append function (in the form of a product), select the second component (ie: `L2`).

When a list being folded over is not empty, the following rewrite rule applies:

$$\begin{aligned}&\langle v,\sigma\rangle;\langle map^{\times}\{P_0,fold^{list}\{g_{nil},g_{cons}\}\},P_1\rangle;g_{cons}\\ \Longrightarrow\;&\langle v,\sigma\rangle;\langle map^{\times}\{P_0,fold^{list}\{g_{nil},g_{cons}\}\},P_1\rangle;\langle P_0;P_0,P_0;P_1\rangle;\mathrm{cons}\end{aligned}$$

5.5. Coinductive Datatypes

Coinductive datatypes are the dual of the inductive datatypes. Computation begins at some initial state with destructors acting on the state to generate a new one. Recall a coinductive definition of the form:

$$\begin{array}{rlcl}\text{data } S \longrightarrow R(A) = & d_1 : S & \longrightarrow & E_1(A,S)\\ | & \vdots & & \\ | & d_n : S & \longrightarrow & E_n(A,S).\end{array}$$

where the state variable lines up on the domain and the codomain contains the type expression. Along with the destructors, three other combinators (record, unfold, map^R) are immediately delivered to the system.

5.5.1. Mapping over coinductive data types

A map over a coinductive datatype has type:

$$map^R\{A_1\times\sigma\to A_1',...,A_m\times\sigma\to A_m'\}:R(A_1,...,A_m)\times\sigma\to R(A_1',...,A_m')$$

The map diagram is shown below:

$$\begin{tikzcd}
R(A) \times \sigma \arrow[r, "d_i \times 1"] \arrow[dd, dashed, "\theta^R"'] & E_i(A, R(A)) \times \sigma \arrow[r, "\langle \theta^{E_i}_{\neg R}, P_1 \rangle"] & E_i(A \times \sigma, R(A)) \times \sigma \arrow[d, "\theta^{E_i}_R"] \\
 & & E_i(A \times \sigma, R(A) \times \sigma) \arrow[d, dashed, "E_i\{I, \theta^R\}"] \\
R(A \times \sigma) \arrow[rr, "d_i"'] & & E_i(A \times \sigma, R(A \times \sigma))
\end{tikzcd}$$

The combinator θ^R strengthens the type R in the same way θ^L strengthens L by propagating the environment through the coinductive data structure. This environment distribution, as shown above, starts from the top left and following the left, bottom path is broken down into the following steps:

(1) apply the destructor d_i to get a term, and propagate the environment,

(2) strengthen the parametric component and propagate the environment,

(3) strengthen the state component,

(4) apply the environment distribution combinator θ^R recursively down the state component.

The rewrite rule for the map^R combinator as derived below also applies the mapped functions to the parametric variables, as well as performing the environment distribution:

$$\begin{aligned}
map^R\{f\}; d_i &\Longrightarrow \theta^R; R\{f\}; d_i \\
&\Longrightarrow d_i \times I; \langle \theta^{E_i}_{\neg R}, P_1 \rangle; \theta^{E_i}_R; E_i\{I, \theta^R\}; E_i\{f, R\{f\}\} \\
&\Longrightarrow d_i \times I; \theta^{E_i}; E_i\{I, \theta^R\}; E_i\{f, R\{f\}\} \\
&\Longrightarrow d_i \times I; \theta^{E_i}; E_i\{f, \theta^R; R\{f\}\} \\
&\Longrightarrow d_i \times I; map^{E_i}\{f, map^R\{f\}\}
\end{aligned}$$

5.5.2. Unfold combinator

The unfold combinator has type:

$$unfold^R\{S \times \sigma \to E_1(A,S), ..., S \times \sigma \to E_n(A,S)\} : S \times \sigma \to R(A)$$

The diagram below shows its recursive nature:

$$\begin{tikzcd}
S \times \sigma \arrow[r, "\langle g_i, P_1 \rangle"] \arrow[dd, dashed, "\text{unfold}^R"'] & E_i(A,S) \times \sigma \arrow[d, "\theta_R^{E_i}"] \\
 & E_i(A, S \times \sigma) \arrow[d, dashed, "E_i\{1, \text{unfold}^R\}"] \\
R(A) \arrow[r, "d_i"] & E_i(A, R(A))
\end{tikzcd}$$

An unfold combinator followed by a destructor on the left, bottom path of the diagram transforms the state S to a new state with the type expected of the destructor. This can be rewritten by following the top, right path as:

(1) Apply the phrase g_i and propagate the environment,

(2) strengthen the state component,

(3) recursively apply the unfold to the state component.

The rewrite rule for the unfold derived from diagram above is:

$$\begin{aligned}
unfold^R\{g_1, ..., g_n\}; d_i &\Longrightarrow \langle g_i, P_1 \rangle; \theta_R^{E_i}; E_i\{I, unfold^R\{g_1, ..., g_n\} \\
&\Longrightarrow \langle g_i, P_1 \rangle; E_i\{P_0, I\}; E_i\{I, unfold^R\{g_1, ..., g_n\} \\
&\Longrightarrow \langle g_i, P_1 \rangle; E_i\{P_0; I, I; unfold^R\{g_1, ..., g_n\}\} \\
&\Longrightarrow \langle g_i, P_1 \rangle; E_i\{P_0, unfold^R\{g_1, ..., g_n\}\}
\end{aligned}$$

5.5.3. Record combinator

The record combinator has type:

$$record^R\{\sigma \to E_1(A, R(A)), ..., \sigma \to E_n(A, R(A))\} : \sigma \to R(A)$$

It allows elements to be added to the head of a coinductive data structure (and it is a non–recursive unfold).

5.6. Translation to combinators

The translation must propagate an environment to the unfold and record combinators. The phrases passed into the unfold and record assume they will have access to it. The translation for the coinductive map operation is exactly the same as over the inductive version.

$$(T_{14}) \quad \mathcal{T}\left[v \mapsto \left(\left| w \mapsto \begin{matrix} d_1 : t_1 \\ \vdots \\ d_n : t_n \end{matrix} \right| \right) (t) \right] = \begin{matrix} \langle \mathcal{T}\left[v \mapsto t\right], 1\rangle ; unfold^R\{ & \mathcal{T}\left[(w, v) \mapsto t_1\right], ..., \\ & \mathcal{T}\left[(w, v) \mapsto t_n\right] \}, \end{matrix}$$

$$(T_{15}) \quad \mathcal{T}\left[v \mapsto \left(\begin{matrix} d_1 : t_1 \\ \vdots \\ d_m : t_m \end{matrix} \right) \right] = record^R\{\mathcal{T}\left[v \mapsto t_1\right], ..., \mathcal{T}\left[v \mapsto t_m\right]\}$$

5.7. Rewrite rules and machine transitions

The co–inductive combinators have the following rewrite rules:

$$\begin{array}{lrcl} (R_{12}) & map^R\{f_1, ..., f_m\}; d_i & \Longrightarrow & d_i \times 1; map^{E_i}\{f_1, ..., f_m, map^R\{f_1, ..., f_m\}\} \\ (R_{13}) & unfold^R\{g_1, ..., g_n\}; d_i & \Longrightarrow & \langle g_i, P_1\rangle; map^{E_i}\{P_0, unfold^R\{g_1, ..., g_n\}\} \\ (R_{14}) & record^R\{h_1, ..., h_n\}; d_i & \Longrightarrow & h_i \end{array}$$

The corresponding state transitions are:

	v	c	$d \vert f$		v'	c'	$d' \vert f'$
22	$map^R\{f_1, ..., f_m\}.v$	$d_i.c$	d	$\longrightarrow$	v	$\langle P_0.d_i, P_1\rangle.\, map^{E_i}\{f_1, ..., f_m,$ $map^R\{f_1, ..., f_m\}\}$	cont$(c).d$
20	$unfold^R\{g_1, ..., g_n\}.v$	$d_i.c$	d	$\longrightarrow$	v	$\langle g_i, P_1\rangle.\, map^{E_i}\{P_0,$ $unfold^R\{g_1, ..., g_n\}\}$	cont$(c).d$
21	$record^R\{h_1, ..., h_n\}.v$	$d_i.c$	d	$\longrightarrow$	v	h_i	cont$(c).d$

5.8. Examples of coinductive combinators

This section provides an example of how coinductive combinators are added to the system.

5.8.1. Streams and infinite lists

A stream is an infinite lists of values, defined as:

```
data C -> stream(A) = head : C -> A
                    | tail : C -> C.
```

The unfold diagram for the stream datatype is:

$$\begin{tikzcd}
C \times \sigma \arrow[d, "\theta_R^{E_i}"'] & Y \times \sigma \arrow[l, "{\langle f, P_1\rangle}"'] \arrow[r, "{\langle g, P_1\rangle}"] \arrow[dd, dashed, "\text{unfold}^R"'] & A \times \sigma \arrow[d, "\theta_R^{E_i}"] \\
C \times \sigma \arrow[d, dashed, "{E_i\{I, \text{unfold}^R\}}"'] & & A \arrow[d, dashed, "{E_i\{I, \text{unfold}^R\}}"] \\
stream(A) & stream(A) \arrow[l, "tail"] \arrow[r, "head"'] & A
\end{tikzcd}$$

The corresponding rewrite rules are:

$$\begin{aligned} unfold^R\{f,g\};head &\Longrightarrow \langle f,P_1\rangle; map^{E_i}\{P_0, unfold^R\{f,g\}\} \\ &\Longrightarrow \langle f,P_1\rangle; P_0 \end{aligned}$$

$$\begin{aligned} unfold^R\{f,g\};tail &\Longrightarrow \langle g,P_1\rangle; map^{E_i}\{P_0, unfold^R\{f,g\}\} \\ &\Longrightarrow \langle g,P_1\rangle; unfold^R\{f,g\} \end{aligned}$$

The *head* will project out the current state, while the *tail* will produce the next state, recursively.

The state transitions delivered by the above rewrite rules are:

v	c	$d\|f'$		v'	c'	$d\|f'$
$unfold^{stream}\{f,g\}.v$	$head.c$	d	$\rightarrow$	v	$\langle f,P_1\rangle; P_0$	$\mathrm{cont}(c).d$
$unfold^{stream}\{f,g\}.v$	$tail.c$	d	$\rightarrow$	v	$\langle g,P_1\rangle; unfold^{stream}\{f,g\}$	$\mathrm{cont}(c).d$

Colists

Colists have the following definition:

```
data S -> colist(A) = delist: S -> sf(A * S).
```

The unfold diagram for `colists` is:

$$\begin{tikzcd}
Y\times\sigma \arrow[r, "\langle f,P_1\rangle"] \arrow[dd, dashed, "\mathrm{unfold}^{colist}\{f\}"'] & sf(A\times Y)\times\sigma \arrow[d, "\theta_R^{E_i}"] \\
 & sf(A\times(Y\times\sigma)) \arrow[d, dashed, "E_i\{I,\mathrm{unfold}^{colist}\{f\}\}"] \\
colist(A) \arrow[r, "delist"'] & sf(A\times Y)
\end{tikzcd}$$

This diagram delivers the rewrite rule for the unfold, where

$$map^{E_i}\{P_0,\mathrm{unfold}^{colist}\{f\}\} = \theta_R^{E_i}; E_i\{I,\mathrm{unfold}^{colist}\{f\}\}$$

is:

$$\begin{aligned} unfold^{colist}\{f\}; delist &\Longrightarrow \langle f, P_1\rangle; map^{E_i}\{P_0, unfold^{colist}\{f\}\} \\ &\Longrightarrow \langle f, P_1\rangle; map^{sf}\{map^{\times}\{P_0, unfold^{colist}\{f\}\}\} \end{aligned}$$

Notice that the combinator map^E generates a map over the datatype `sf`.

Using the equation for map^R, the map^{colist} combinator is defined as:

$$\begin{aligned} map^{colist}\{f\}; delist &\Longrightarrow \langle delist, I\rangle; map^{E_i}\{f, map^{colist}\{f\}\} \\ &\Longrightarrow \langle delist, I\rangle; map^{sf}\{map^{\times}\{f, map^{colist}\{f\}\}\} \end{aligned}$$

The state transition rules are:

v	c	$d\|f'$		v'	c'	$d\|f'$
$unfold^{colist}\{f\}.v$	$delist.c$	d	$\rightarrow$	v	$\langle f, P_1\rangle; map^{sf}\{map^{\times}\{P_0,$ $unfold^{colist}\{f\}\}\}$	$\mathrm{cont}(c).d$
$map^{colist}\{f\}.v$	$delist.c$	d	$\rightarrow$	v	$\langle f, I\rangle; map^{sf}\{f, map^{colist}\{f\}\}$	$\mathrm{cont}(c).d$
$record^{colist}\{f\}.v$	$delist.c$	d	$\rightarrow$	v	f	$\mathrm{cont}(c).d$

Both the unfold and map operations over colists must perform a map over the inductive `sf` datatype to generate the next state. If the `ff` constructor is returned, the computation finishes, while a `ss` causes a recursion to generate the next state.

Chapter 6

Implementing The Charity Abstract Machine

The by–value Charity Abstract Machine described in the last section is a prototype machine focusing on simplicity and correctness rather than efficient use of computer resources. An improved implementation of this abstract machine should aim to improve its execution speed and memory usage, while retaining its correctness. The first part of this chapter highlights those drawbacks of the basic version which influenced the design of a new abstract machine and compiler stage.

The Charity Abstract Machine presented in the last chapter has several shortcomings:

- There is no sharing of subexpressions through lazy graph reduction. Subexpressions may be referenced more than once, but with no sharing mechanism, the code for the expression is duplicated and may be evaluated multiple times.
- Code is generated at run time. Similar to a simple template instantiation machine, the template for a function must be traversed at run time and arguments substituted within in the body of the function. For example, the map^{E_i} combinator must generate code for an operation on a datatype, forcing the machine to suspend execution while this code is created.

6.1. Designing a new abstract machine

The new abstract machine will not execute combinators directly. Instead a simple *macro–code* language, closer to a physical machine instruction set is used. The categorical combinators are compiled to this macro–code language. The compiler presented in this chapter deals more with efficient code generation, rather than optimization and partial evaluation of the combinators. The resultant machine code instructions are simple and mechanical macro instructions. Much of the pattern matching associated with the by–value machine is replaced with more straight forward assembly–like instructions. Furthermore, a more uniform representation of data has been adopted.

6.1.1. The abstract machine

The target abstract machine is described in tables similar to the original machine. The macro machine contains four stacks as before:

- v value stack,
- c code stack (contains *macro* instructions),
- d dump stack, and
- f function stack

In addition, this machine contains a heap store h where a large storage space is set aside to hold uniform data values generated by the abstract machine. Each heap item is tagged with a value to distinguish heap items from each other. The only permissible values in the heap are:

- $()$: an object of type 1
- a product $\langle h_0, h_1\rangle$
- a constructor CONS$\{i\}$
- a record $(r_0, ..., r_n)$

where h_0, h_1, r_0, and r_n, are heap values. The layout of the state transition tables is slightly different from the machine in the previous chapters. The main difference is that an item listed on the value stack is really a pointer to one heap location (the root of a tree structure in the heap). Such a pointer is a *primary* pointer into the heap. Other references to heap items are listed below the primary pointer as $v : x$ which says "a pointer v points to a heap location with contents x". For example, the state transition:

	$v \mid h$	c	$d \mid f$	$\rightarrow$	$v' \mid h'$	c'	$d' \mid f'$
11	$v : \langle v_0, v_1 \rangle$ $v_0 : c_i.v_0'$	INDUCT$\{f_i\}_{i=1}^{n}.c$	d	$\rightarrow$	$v' : \langle v_0', v_1 \rangle$	f_i	cont$(c).d$

is read as "v is a pointer to a heap item $\langle v_0, v_1 \rangle$, and the v_0 is also a pointer to a constructor c_i also located in the heap". If the pointer does not point to anything of relevance for that state transition, the just pointer appears as the variable name.

The basic state transitions listed in table 6.1 describe the formation of pairs, projection on pairs and cases where the code stack is empty. These transitions are essentially the same as the state transitions for the basic combinators in the Charity abstract machine of chapter 4. The main difference lies in the evaluation of the pair combinator. Instead of evaluating pairs starting with the first component, then doing second component, the macro machine evaluates the second component before the first. The machine optimistically assumes that the second component of a pair just reproduces environment (eg: contains only a single identity combinator). In this case, the environment is distributed to the second component and no combinator code needs to be executed. The reason for the choice in evaluation of a product will become apparent with the introduction of the datatypes.

	$v \mid h$	c	$d \mid f$	$\rightarrow$	$v' \mid h'$	c'	$d' \mid f'$
1	v	$!.c$	d	$\rightarrow$	$!$	c	d
2	$v : \langle v_0, v_1 \rangle$	$P_0.c$	d	$\rightarrow$	v_0	c	d
3	$v : \langle v_0, v_1 \rangle$	$P_1.c$	d	$\rightarrow$	v_1	c	d
4	v	$\langle c_0, c_1 \rangle.c$	d	$\rightarrow$	v	c_1	$\mathrm{pr}_0(v, c_0, c).d$
5	v_1	ϵ	$\mathrm{pr}_0(v, c_0, c).d$	$\rightarrow$	v	c_0	$\mathrm{pr}_1(v_1, c).d$
6	v_0	ϵ	$\mathrm{pr}_1(v_1, c).d$	$\rightarrow$	$v : \langle v_0, v_1 \rangle$	c	d
7	v	ϵ	$\mathrm{cont}(c).d$	$\rightarrow$	v	c	d
8	v	ϵ	$[\,]$	$\rightarrow$	HALT		

Table 6.1. State transition rules for the basic instructions

6.1.2. Sharing in products

When a value is distributed over a product, the value distributed is shared among the components of the product. This is in direct contrast to the by–value machine where a shared combinator is copied, then distributed to the components. Sharing in this machine is accomplished by using pointers to reference shared values, instead of copying the entire structure. For products, the pointer v in state transition 4 is pushed onto the dump and reloaded onto the value stack by transition 5. Thus both state transition rules use the distributed value, but no copying is required. For example, consider the diagonal $x; \langle I, I \rangle$. Distributing x over the pair should rewrite to $\langle x; I, x; I \rangle = \langle x, x \rangle$. What really happens is the pointer x is reproduced and not the value x points to.

6.1.3. Mapping over products and the type 1

Mapping over the basic type 1 has the same action as the P_0 macro instruction, and so will produce the same code as the P_0. However mapping over products ($\times$) requires a new macro instruction with the state transition as shown in table 6.2. The $map^{\times}$ instruction performs the same function as the $map^{\times}$ combinator described in

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
9	$v : \langle v', v_2 \rangle$ $v' : \langle v_0, v_1 \rangle$	$\mathrm{MAP}^{\times}\{f, g\}.c$	d	$\rightarrow$	$v'' : \langle v_1, v_2 \rangle$ $v'_0 : \langle v_0, v_2 \rangle$	g	$\mathrm{pr}_0(v'_0, f, c).d$

TABLE 6.2. State transition rule for mapping over products

the previous chapter.

6.1.4. Compiling categorical combinators

straightforward compilation from categorical combinators to macro–instructions is given by:

(C_1) $\mathcal{C}[!] = !,$

(C_2) $\mathcal{C}[\langle f, g \rangle] = \langle \mathcal{C}[f], \mathcal{C}[g] \rangle,$

(C_3) $\mathcal{C}[f; g] = \mathcal{C}[f].\mathcal{C}[g],$

(C_4) $\mathcal{C}[p_i] = P_i,$ where $i = 0, 1,$

(C_5) $\mathcal{C}[I] = \epsilon,$ where ϵ is the empty code stream

(C_6) $\mathcal{C}[map^{\times}\{f, g\}] = \mathrm{MAP}^{\times}\{\mathcal{C}[f], \mathcal{C}[g]\}$

(C_7) $\mathcal{C}[map^{1}\{\}] = P_0$

Composition of combinators in C_3 is compiled into a sequence of code for f followed by code for g. The identity combinator in C_5 does not produce any code.

6.2. Inductive Datatypes

Datatypes in the new machine are handled in a more uniform, simple manner than in the original machine. Inductive datatypes with their corresponding fold, map and case combinators are generalized into a single INDUCT instruction. This INDUCT instruction holds pointers to the compiled code for each constructor in the datatype as shown in figure 6.1. The c_1 to c_n represent the compiled code for each constructor of the datatype, and applying a constructor to an INDUCT instruction will simply choose the appropriate code to execute.

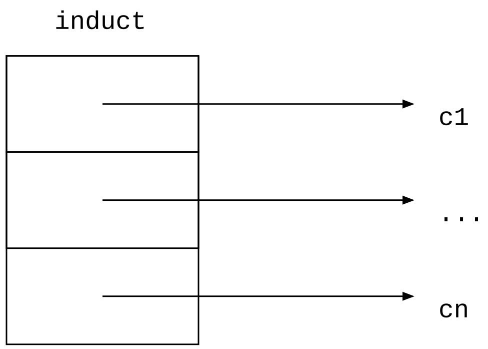

FIGURE 6.1. The inductive combinator

Thus this generalized INDUCT instruction provides a more uniform representation and more mechanical instruction processing leading to considerable simplification of the machine. Evaluating a inductive operator amounts to chasing a pointer to some compiled code that inherently knows what operation it is. The complexity of the machine is reduced (as it should be) at the cost of slightly increasing the complexity of the compiler.

A further benefit of creating a generic macro combinator for the inductive datatype operations is that it enables the elimination of the code generation required by the map^{E_i} combinator at run time. Instead, the compiler is given the responsibility for producing code for any application of the induct combinator to a constructor.

6.2.1. Compiling inductive combinators

Constructors and the inductive combinators fold, case and map are compiled with:

$$\begin{array}{rlcl}
(C_8) & \mathcal{C}[c_i] & = & \mathrm{CONS}\{i\}, \\
(C_9) & \mathcal{C}[case^L\{f_1, ..., f_n\}] & = & \mathrm{INDUCT}\{f_i'\}_{i=1}^n \\
 & & & \text{where } f_i' := \mathcal{C}[f_i], \\
(C_{10}) & \mathcal{C}[fold^L\{g_1, ..., g_n\}] & = & \mathrm{INDUCT}\{g_i'\}_{i=1}^n \\
 & & & \text{where } g_i' := \langle \mathcal{C}[\mathrm{map}^{E_i}\{p_0, \mathrm{induct}\{g_i'\}_{i=1}^n\}], P_1 \rangle . \mathcal{C}[g_i], \\
(C_{11}) & \mathcal{C}[map^L\{h_1, ..., h_m\}] & = & \mathrm{INDUCT}\{h_i'\}_{i=1}^n \\
 & & & \text{where } h_i' := \mathcal{C}[\mathrm{map}^{E_i}\{h_1, ..., h_m, \mathrm{induct}\{h_i'\}_{i=1}^n\}].\mathrm{CONS}\{i\}
\end{array}$$

A new combinator $induct\{h_i'\}_{i=1}^n$ is introduced to temporarily hold the inductive instructions. The Compilation of this instruction produces the INDUCT macro instruction:

$$(C_{12}) \quad \mathcal{C}[induct\{h_i'\}_{i=1}^n] \quad = \quad \mathrm{INDUCT}\{h_i'\}_{i=1}^n$$

At run time, the induct combinator does not know what constructor it will encounter, and therefore must assume that any constructor of the datatype could be encountered. The code for each constructor applied to the induct combinator is compiled and stored as an offset in the induct combinator itself. An application of the ith constructor will cause the ith compiled code to be loaded and executed.

Finally, the state transitions of the inductive combinators follows:

	$v\|h$	c	$d\|f$	→	$v'\|h'$	c'	$d'\|f'$
10	v	$\mathrm{CONS}\{i\}.c$	d	→	$u : \mathrm{cons}\{i\}.v$	c	d
11	$v : \langle v_0, \sigma \rangle$	$\mathrm{INDUCT}^L\{c_i\}_{i=1}^n.c$	d	→	$v' : \langle v_0', \sigma \rangle$	c_i	$\mathrm{cont}(c).d$
	$v_0 : \mathrm{cons}\{i\}.v_0'$						

TABLE 6.3. State transition rules for the inductive datatypes

6.2.2. Partial evaluation of inductive operators

Upon reexamining the execution of the fold combinator, partial evaluation can be applied to the expression. Since the fold produces a pair, reproducing the environment in the second component, the machine could avoid evaluating the second component and reproduce the environment explicitly. To achieve this short cut, the machine uses the instruction PR. Normally, propagating the environment in a fold is accomplished with the following rewrite:

$$\langle v, \sigma\rangle ; \langle f, P_1\rangle \rightarrow \langle\langle v, \sigma\rangle ; f, \sigma\rangle$$

which distributes $\langle u, \sigma\rangle$ over the second pair and passes along the environment. The rewrite could be improved by replacing the pairing instruction with the PR instruction, eg:

$$\langle v, \sigma\rangle ; \langle f, P_1\rangle \Longrightarrow \langle v, \sigma\rangle ; \mathrm{PR}\{f\}$$

As a result of this instruction, the fold combinator is compiled slightly differently to take advantage of this optimization.

$$\begin{array}{llcl} (C'_{10}) & \mathcal{C}[fold^L\{g_1, ..., g_n\}] & = & \mathrm{INDUCT}\{g'_i\}_{i=1}^n \\ & & & \text{where } g'_i := \mathrm{PR}\{\mathcal{C}[\mathrm{map}^{E_i}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^n\}]\}.\mathcal{C}[g_i], \end{array}$$

The state transition for the PR instruction is:

	$v \vert h$	c	$d \vert f$	$\rightarrow$	$v' \vert h'$	c'	$d' \vert f'$
12	$v : \langle v_0, \sigma\rangle$	$\mathrm{PR}\{c'\}.c$	d	$\rightarrow$	$v : \langle v_0, \sigma\rangle$	c'	$\mathrm{pr}_1(\sigma, c).d$

TABLE 6.4. Additional state transition rules for datatypes

When the environment only needs to be duplicated (or passed on as in the fold instruction), there is no code to execute in the second component. Instead, the environment being distributed over the pair is pushed onto the dump and the first component of the pair evaluated. When evaluation of the first component reaches normal form, a pair is constructed on the heap (by state transition 5), with v pointing to the newly constructed value.

6.2.3. Example: Compiling the append function

The append function is translated to the following combinator expression:

$$\langle p_0, I\rangle; \mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}$$

To compile this combinator expression, we use the $\mathcal{C}$ compilation scheme.

$$\begin{array}{rl}
 & \mathcal{C}[\langle p_0, I\rangle; \mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}] \\
\Longrightarrow & \mathcal{C}[\langle p_0, I\rangle].\mathcal{C}[\mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}] \\
\Longrightarrow & \langle \mathcal{C}[p_0], \mathcal{C}[I]\rangle.\mathcal{C}[\mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}] \\
\Longrightarrow & \langle P_0, \mathcal{C}[I]\rangle.\mathcal{C}[\mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}] \\
\Longrightarrow & \langle P_0, \epsilon\rangle.\mathcal{C}[\mathit{fold}^{\text{list}}\{p_1; p_1, \langle p_0; p_0, p_0; p_1\rangle; \text{cons}\}] \\
\Longrightarrow & \langle P_0, \epsilon\rangle.\text{INDUCT}\{g_i'\}_{i=1}^{2}
\end{array}$$

where

$$\begin{aligned}
g'_1 \quad &:= \quad \mathrm{PR}\{\mathcal{C}[\mathrm{map}^{E_1}\{p_0, \mathrm{induct}\{g'_i\}_{i=1}^{2}\}]]\}.\mathcal{C}[p_1; p_1] \\
&\Rightarrow \quad \mathrm{PR}\{\mathcal{C}[map^1\{\}\}]]\}.\mathcal{C}[p_1; p_1] \\
&\Rightarrow \quad \mathrm{PR}\{P_0\}.\mathcal{C}[p_1; p_1] \\
&\Rightarrow \quad \mathrm{PR}\{P_0\}.\mathcal{C}[p_1].\mathcal{C}[p_1] \\
&\Rightarrow \quad \mathrm{PR}\{P_0\}.P_1.\mathcal{C}[p_1] \\
&\Rightarrow \quad \mathrm{PR}\{P_0\}.P_1.P_1
\end{aligned}$$

$$\begin{aligned}
g'_2 \quad &:= \quad \mathrm{PR}\{\mathcal{C}[map^{E_2}\{p_0, \mathcal{C}[induct\{g'_i\}_{i=1}^{2}]\}]]\}.\mathcal{C}[\langle p_0; p_0, p_0; p_1\rangle; \mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{\mathcal{C}[p_0], \mathcal{C}[induct\{g'_i\}_{i=1}^{2}]\}\}.\mathcal{C}[\langle p_0; p_0, p_0; p_1\rangle; \mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathcal{C}[induct\{g'_i\}_{i=1}^{2}]\}\}.\mathcal{C}[\langle p_0; p_0, p_0; p_1\rangle; \mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\mathcal{C}[\langle p_0; p_0, p_0; p_1\rangle; \mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\mathcal{C}[\langle p_0; p_0, p_0; p_1\rangle].\mathcal{C}[\mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\langle \mathcal{C}[p_0; p_0], \mathcal{C}[p_0; p_1]\rangle.\mathcal{C}[\mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\langle \mathcal{C}[p_0].\mathcal{C}[p_0], \mathcal{C}[p_0; p_1]\rangle.\mathcal{C}[\mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\langle \mathcal{C}[p_0].\mathcal{C}[p_0], \mathcal{C}[p_0].\mathcal{C}[p_1]\rangle.\mathcal{C}[\mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\langle P_0.P_0, P_0.P_1\rangle.\mathcal{C}[\mathrm{cons}] \\
&\Rightarrow \quad \mathrm{PR}\{\mathrm{MAP}^{\times}\{P_0, \mathrm{INDUCT}\{g'_i\}_{i=1}^{2}\}\}.\langle P_0.P_0, P_0.P_1\rangle.\mathrm{CONS}(2)
\end{aligned}$$

6.3. Coinductive Datatypes

Similar to inductive datatypes, coinductive datatypes with their unfold, map, and record combinators are generalized into the RECORD instruction which holds pointers to code for each destructor in the datatype. An application of a destructor to the coinduct datatype loads and executes the proper code.

6.3.1. Lazy coinductive datatypes

Coinductive datatypes are evaluated in a by–need manner as opposed to the by–value mechanism used in the original machine. Lazy evaluation adopts the philosophy of "do the minimum amount of work possible" in an attempt to eliminate unnecessary computation. Lazy evaluation in the Charity abstract machine means that a coinductive operator (unfold, map or record) is not evaluated until a destructor is applied to the operator. The abstract machine of the previous chapter simply pushed the *unfold*, *map*, or *record* combinator onto the value stack and continued processing the sequence of code. Destructors applied to this operator execute a piece of code stored in the RECORD instruction.

The new machine requires additional machinery to deal with the lazy evaluation. Specifically, a record is created to hold *closures*. A closure holds a piece of code that is potentially unevaluated. It is represented by the pair

$$(v, c)$$

where v is the value acted on by some code c. A record structure holds the list of closures for each destructor of a coinductive datatype, and is defined as:

$$u : \text{rec}\{clo[i] : (v, c)\}_{i=1}^{n}$$

where u points to the record in the heap and $clo[i]$ is an index into the ith of n closures. Applying a destructor to a record will load, or *enter* the closure to be evaluated. Upon exit of the closure, the value has been computed and can be used in a later computation. The state transition (13) for the record combinator simply creates a record of closures on the value stack:

	$v \vert h$	c	$d \vert f$	$\rightarrow$	$v' \vert h'$	c'	$d' \vert f'$
13	v	$\text{RECORD}\{f_i\}_{i=1}^{n}.c$	d	$\rightarrow$	$v' : \text{rec}\{clo[i]\}_{i=1}^{n}$ $clo[i] : (v, f_i)$	c	d

TABLE 6.5. State transition rules for the records

All coinductive operators are compiled into the RECORD instruction. The difference lies in the code generated for each destructor phrase of the record.

$$\begin{array}{lll}
(C_{13}) & \mathcal{C}[unfold^R\{f_1, ..., f_n\}] & = \text{RECORD}\{f'_i\}_{i=1}^n \\
 & & \text{where } f'_i = \langle \mathcal{C}[f_i], P_1\rangle.\mathcal{C}[\text{map}^{E_i}\{p_0, record\{f'_i\}_{i=1}^n\}], \\
(C_{14}) & \mathcal{C}[record^R\{g_1, ..., g_n\}] & = \text{RECORD}\{\mathcal{C}[g_1], ..., \mathcal{C}[g_n]\}, \\
(C_{15}) & \mathcal{C}[map^R\{h_1, ..., h_m\}] & = \text{RECORD}\{h'_i\}_{i=1}^m \\
 & & \text{where } h'_i = \langle P_0.\text{DESTR}\{i\}, P_1\rangle \\
 & & \quad .\mathcal{C}[\text{map}^{E_i}\{h_1, ..., h_m, record\{h'_i\}_{i=1}^m\}]
\end{array}$$

As with compiling of the inductive operations, a intermediate combinator *record* is used for recursive calls. Compiling this combinator is done by the following:

$$(C_{16}) \quad \mathcal{C}[record\{g_1, ..., g_n\}] \quad = \quad \text{RECORD}_{i=1}^n$$

6.3.2. Adding sharing to coinductive datatypes

When evaluating coinductive combinators, lazy evaluation guarantees the machine will not perform unnecessary computation. For example, a destructor may be applied to a record, resulting in an entry into a closure to produce an answer. Now consider the consequence of applying the same destructor to the same record, which then results in the evaluation of the same closure. Thus, one computes the same answer as in the previous application. However, unnecessary computation has been eliminated since only the answer is returned and the code is not executed twice.

To ensure that the code in a closure is executed at most once, a sharing mechanism needs to be incorporated into the machine. The prime component of the sharing mechanism is the updating of closures with their results, as they exit. The machine pushes an update marker on top of the dump stack to tag the closure that must be updated. When the machine exits the closure, the top of the dump marks the closure that must be overwritten and updated with the result. The code portion of this closure is also overwritten with an empty sequence of code (ϵ), such that future entries into this closure will simply return the value previously computed. Thus, the resultant value of entering a closure is shared among multiple references to that closure.

State transition 14 applies a destructor to the record, while state transition 15 updates a closure upon exit. To update a closure, state transition 15 overwrites the closure node in the heap with the value computed, along with an empty code stream for the code in the closure.

	$v\|h$	c	$d\|f$		$v'\|h'$	c'	$d'\|f'$
14	$v : \mathrm{rec}\{v[i]\}_{i=1}^{n}$	$\mathrm{DESTR}\{i\}.c$	d	$\longrightarrow$	v_i	f_i	$\mathrm{update}(clo[i]).$
	$clo[i] : (v_i, f_i)$						$\mathrm{cont}(c).d$
15	v	ϵ	$\mathrm{update}(clo[i]).d$	$\longrightarrow$	v	ϵ	d
	$clo[i] : (v_i, f_i)$				$clo[i] : (v, \epsilon)$		

TABLE 6.6. State transition rules for the destructors and updating a closure

Similar to the fold on inductive datatypes, the compilation for the unfold and map combinators on coinductive datatypes can take advantage of environment passing using the PR instruction:

$$\begin{aligned}
(C'_{13}) \quad & \mathcal{C}[unfold^{R}\{f_1, ..., f_n\}] &=\ & \mathrm{RECORD}\{f'_i\} \\
& & & \text{where } f'_i = \mathrm{PR}\{\mathcal{C}[f_i]\}.\mathcal{C}[\mathrm{map}^{E_i}\{P_0, record\{f'_i\}\}] \\
(C'_{15}) \quad & \mathcal{C}[map^{R}\{h_1, ..., h_m\}] &=\ & \mathrm{RECORD}\{h'_i\} \\
& & & \text{where } h'_i = \mathrm{PR}\{P_0.\mathrm{DESTR}\{i\}\}.\mathcal{C}[\mathrm{map}^{E_i}\{h_1, ..., h_m, record\{h'_i\}\}]
\end{aligned}$$

6.4. Adding the Natural numbers as primitive types

Since considerable processing in Charity is done using some bounded computation, natural numbers are added as a primitive type to improve performance. The tail recursion associated with a fold on the natural numbers can be replaced with a “for” loop construct. In addition, the storage requirements of the primitive natural numbers is considerably reduced. There is no longer a need to create a linked list of boxed unary constructors. Instead, a natural number object is introduced to hold the integer value of the natural number.

State transition 16 puts a natural number object in the heap. The natural number object stores numbers between 0 and n. The instruction "NAT$\{n\}$" builds a single natural number heap item as shown by state transition 16. Adding one to a natural number simply increases the value of the natural number item in the heap by one. State transition 17 shows how the instruction "SNAT" adds one to a natural number heap item. Notice that only one heap item is required to represent a natural number with value n instead of n heap items as required in a unary representation.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
16	v	$\mathrm{NAT}\{n\}.c$	d	$\rightarrow$	$v' : \mathrm{nat}\{n\}.v$	c	d
17	$\mathrm{nat}\{n\}$	$\mathrm{SNAT}.c$	d	$\rightarrow$	$\mathrm{nat}\{n+1\}$	c	d

TABLE 6.7. State transition for $\mathcal{N}$ datatype

The CASE_NAT instruction over the natural numbers acts in the same manner as a CASE instruction on any datatype as shown by state transitions 18 and 19. However, the case over natural numbers must be careful to act on the special natural number element in the heap.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
18	$v : \langle v_0, \sigma\rangle$ $v_0 : \mathrm{nat}\{z\}.v_0'$	$\mathrm{CASE\_NAT}\{f_z, f_s\}.c$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	f_z	$\mathrm{cont}(c).d$
19	$v : \langle v_0, \sigma\rangle$ $v_0 : \mathrm{nat}\{s(n)\}.v_0'$	$\mathrm{CASE\_NAT}\{f_z, f_s\}.c$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	f_s	$\mathrm{cont}(c).d$

TABLE 6.8. State transition for $\mathcal{N}$ "case" instruction

The fold over the natural numbers removes tail recursion by transforming the fold combinator into a "for" loop as in:

```
initialize loop
for i := 1 to n do
  ...
```

Loop setup occurs by executing the "zero" phrase to set the initial value. Next, an iteration occurs where the "succ" phrase is repeated n times. A new dump item *loop* is the loop counter, beginning at the number n. In addition, *loop* holds the environment σ and the "succ" phrase to be executed at each iteration. State transition 20 initializes the machine in preparation for the loop. A test of whether to iterate or terminate the loop is made in state transitions 21 and 22.

	$v \mid h$	c	$d \mid f$	$\rightarrow$	$v' \mid h'$	c'	$d' \mid f'$
20	$v : \langle v_0, \sigma\rangle$ $v_0 : \text{nat}\{n\}.v'_0$	FOLD_NAT$\{f_z, f_s\}.c$	d	$\rightarrow$	$v' : \langle v'_0, \sigma\rangle$	f_z	$\text{loop}(n, \sigma, f_s).$ $\text{cont}(c).d$
21	v	ϵ	$\text{loop}(s(n), \sigma, f_s).d$	$\rightarrow$	$v' : \langle v, v_2\rangle$	f_s	$\text{loop}(n, \sigma, f_s).d$
22	v	ϵ	$\text{loop}(z, \sigma, f_s).d$	$\rightarrow$	$\langle v, \sigma\rangle$	ϵ	d

TABLE 6.9. State transition for $\mathcal{N}$ "fold" instruction

6.5. Displaying coinductive datatypes

When displaying a record the result of a computation, only one level of the record is displayed, as the record may be infinite. For example, the infinite list of natural numbers is the record:

$$(head : 0, tail : (head : 1, tail : (head : 2, tail : ...)))$$

The infinite nature of the tail component makes it impossible to show the entire record. Instead, only one level of the infinite list is displayed at one time. In the example above, the record

$$(head : ..., tail : ...)$$

would be displayed, until prompted for the next tail item to produce:

$$(head : 0, tail : ...)$$

then one could prompt again:

$$(head : 0, tail : (head : ..., tail : ...))$$

until the user quits from what is called the "coinductive display mode". This mode is useful for viewing portions of an element of a coinductive datatype incrementally.

To implement the coinductive display, the machine is initially loaded (as usual) with the code that builds the record. When computation has completed, the value stack will be in head normal form, holding a record.

$v \mid h$	c	$d \mid f$
$v : \text{rec}\{clo[i]\}_{i=1}^{n}$	ϵ	$[\,]$
$clo[i] : (v_i, c_i)$		

To generate the next value (ie: to look one level down), we would apply each destructor for R to the record:

$v \mid h$	c	$d \mid f$
$v : \text{rec}\{clo[i]\}_{i=1}^{n}$	d_i	$[\,]$

The consequence of this action is that each closure in the record is entered (and evaluated) by loading the machine with the value and code of each closure in succession. For example, to enter the ith closure, the machine state will start as:

$v \mid h$	c	$d \mid f$
v_i	c_i	$\text{update}(rec, i).d$

When the record completes computation, the original closure is updated with the value and an empty code stream, leaving the state of the machine as:

$v \mid h$	c	$d \mid f$
v'	ϵ	$[\,]$
$\mathrm{rec}\{clo[i]\}_{i=1}^{n}$		
$clo[i] : (v'_i, \epsilon)$		

Chapter 7
Linearizing the abstract machine

Optimizations of the machine can be realized by making the state transitions linear. That is, take out the places where the machine must examine the top of each stack to make a decision about the code to execute. Instead, the instruction on the code stream determines the state transition of the machine. The sequence of code then executes in a more linear fashion, where execution starts from an actual "beginning" and continues to the "end". This linear set of instructions is more amenable to translation down to native machine code.

The design philosophy of the linear abstract machine consists of the following principles:

(1) Like the previous machines, code is never stored in the value stack. But unlike the previous machines, the code stack is never temporarily empty. The machine will never have to inspect the dump stack for more code once the code stack empties itself. That is, the dump stack may contain pointers to code, but the machine never executes the code in the dump directly. Rather, a "jump" is initiated to load the program counter with the appropriate pointer to the next piece of code to execute.

(2) Macro instructions of the previous machine are replaced with even smaller linear micro instructions. Instead of having one macro instruction perform many small tasks, we choose to split macro instruction into several smaller instructions that, in combination, carry out the work of the macro instruction. An

example is the contrast between how pairs are built in the previous machine, versus in the new machine.

7.1. Basic combinators

The basic combinators in this machine are composition, terminal map, identity and pairs. The identity combinator does not generate any code, while the ! combinator generates the micro instruction !. Composition of combinators is reduced to generating a sequence of code in a linear order. Now we come to our first linearization combinators for products. This linearization is basically the that of Curien's Categorical Abstract Machine [Cur86]. Distributing a value over pair combinator with

$$v.\langle f, g\rangle = \langle v.f, v.g\rangle$$

is divided into three instructions: SAVE, SWAP and a new PAIR instruction. SAVE simply pushes the v on the value stack to be distributed onto the dump. SWAP reloads the saved v by swapping the current value of v and the value pushed onto the dump. Finally, the PAIR instruction constructs a pair on the value stack. To evaluate the first and second component of a pair, we simply compile the code for the second component between the SAVE and SWAP instructions, and compile the first component between the SWAP and PAIR instructions. Note that in compiling a pair we choose to compile (and evaluate) the second component ahead of the first. Like the previous machine, the compiler performs some partial evaluation where the machine optimistically assumes that the environment is in the second component. Thus when the environment needs to be replicated, the machine uses the PR instruction to automatically project it out and push it onto the stack.

The compilation scheme for these combinators are:

(C_1) $\mathcal{C}[c_0; c_1] = \mathcal{C}[c_0].\mathcal{C}[c_1]$,

(C_2) $\mathcal{C}[!] = !$,

(C_3) $\mathcal{C}[I] = \epsilon$ where ϵ is the empty code stream,

(C_4) $\mathcal{C}[p_i] = P_i$ for $i = 0, 1$,

(C_5) $\mathcal{C}[\langle c_0, c_1 \rangle] = \mathrm{SAVE}.\mathcal{C}[c_1].\mathrm{SWAP}.\mathcal{C}[c_0].\mathrm{PAIR}$,

(C_6) $\mathcal{C}[map^1\{\}] = P_0$,

(C_7) $\mathcal{C}[map^\times\{f, g\}] = \mathrm{MAP}^\times.\mathcal{C}[f].\mathrm{SWAP}.\mathcal{C}[g].\mathrm{PAIR}$

The sequence in which a pair combinator will execute as follows:

(1) copy a value onto the dump for distribution over a pair,

(2) execute the compiled code associated with the second component of the pair,

(3) swap the resulting value with the dump,

(4) execute the compiled code associated with the first component of the pair,

(5) recombine the item on the top of the dump stack with the current item on the value stack to create a pair combinator to be left on the value stack.

The state transition rules of table 7.1 show how the first three transitions retain the same operation as in the previous machine. However, transitions 4 to 6 work in conjunction to perform the task of the pair combinator. Transition 7 is the map over products, while the PR instruction in transition 8 allows for the environment to be reproduced with minimal waste in instructions. An explicit HALT instruction in transition 9 stops the machine. This is in contrast to checking the code and dump stacks to make sure they are both empty before halting. The jump transition (10) is analogous to a subroutine call where the machine "jumps" to the a piece of code and continues execution there. When a RET (transition 11) instruction is encountered, the suspended sequence of code pushed onto the dump stack is popped off and execution resumes as before. Finally, the machine can execute an unconditional goto to execute some other piece of code, as in transition 12.

	$v \mid h$	c	$d \mid f$	$\rightarrow$	$v' \mid h'$	c'	$d' \mid f'$
1	v	$!.c$	d	$\rightarrow$	$!$	c	d
2	$v : \langle v_0, v_1 \rangle$	$P_0.c$	d	$\rightarrow$	v_0	c	d
3	$v : \langle v_0, v_1 \rangle$	$P_1.c$	d	$\rightarrow$	v_1	c	d
4	v	SAVE.c	d	$\rightarrow$	v	c	$\mathrm{pr}_0(v).d$
5	v_1	SWAP.c	$\mathrm{pr}_0(v).d$	$\rightarrow$	v	c	$\mathrm{pr}_1(v_1).d$
6	v_0	PAIR.c	$\mathrm{pr}_1(v_1).d$	$\rightarrow$	$v : \langle v_0, v_1 \rangle$	c	d
7	$v : \langle v', \sigma \rangle$ $v' : \langle v_0, v_1 \rangle$	MAP$^{\times}$.c	d	$\rightarrow$	$v'' : \langle v_1, \sigma \rangle$	c	$\mathrm{pr}_0(\langle v_0, \sigma \rangle).d$
8	$v : \langle v_0, v_1 \rangle$	PR.c	d	$\rightarrow$	$v : \langle v_0, v_1 \rangle$	c	$\mathrm{pr}_1(v_1).d$
9	v	HALT	[ ]	$\rightarrow$		STOP	
10	v	JUMP$\{c'\}.c$	d	$\rightarrow$	v	c'	$\mathrm{cont}(c).d$
11	v	RET	$\mathrm{cont}(c).d$	$\rightarrow$	v	c	d
12	v	GOTO$\{c\}$	d	$\rightarrow$	v	c	d

TABLE 7.1. State transition rules for the basic instructions

7.2. The linear datatypes

Moving towards a linear machine makes the overall operation of the machine much simpler. The combinators for the datatypes benefit from this linear machine by a more efficient and uniform code representation. All instructions contain at most one operand. This operand is either a pointer to some code, a pointer to an offset table, or an integer value. Applying a constructor to an inductive combinator simply enters the code associated with that constructor.

In making the datatypes linear, we must reexamine the combinators generated by the inductive and coinductive datatypes. The INDUCT, RECORD and DESTR combinators that performed many tasks in the previous machine now are modified to perform at most one task. By task, we mean a single simple stack operation or reference. The design philosophy is to make the machine less abstract and more concrete. Instructions are now similar to physical machine instructions, making the gap between abstract machine and machine code generation a small leap.

To achieve speeds comparable to imperative languages, functional languages are compiled to native machine code. Whether this compilation is the result of compiling to some high level then to machine code, or directly to machine code, the goal of most implementations is to choose an appropriate intermediate language.

7.3. Linear inductive datatypes

The INDUCT combinator of the last chapter is modified into a linear instruction where this combinator points to the start of a code table. Each entry in this table is a goto instruction that points to code compiled for one of the inductive operators (case, fold, or map), one GOTO for each constructor. For example, if the INDUCT combinator is a fold, then a constructor CONS{i} applied to this INDUCT combinator would select the ith code to execute from the offset table. The code for the

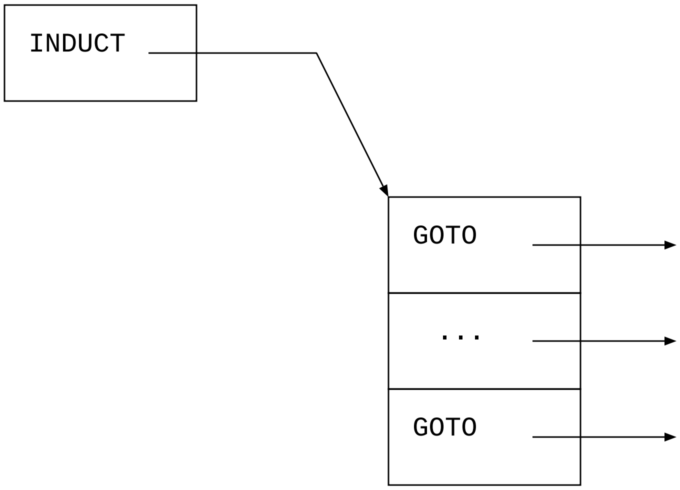

FIGURE 7.1. A view of the induct instruction

case under CONS$\{i\}$ has been compiled to the appropriate linear instructions. The INDUCT instruction is, in essence, a conditional indirection pointer and points to a table of GOTO instructions. The compiler is given the task of compiling the map, fold, or case combinator and must take into account the effect of each constructor on the inductive datatype. The net effect is that the induct combinator no longer needs to know about a particular fold, map, or case combinator.

The translation $\mathcal{C}$ from the categorical combinators is described as follows:

$$\begin{array}{lll}
(C_8) & \mathcal{C}[c_i] & = \text{CONS}(i), \\
(C_9) & \mathcal{C}[\text{case}^L\{f_0,...,f_n\}] & = x := \text{INDUCT}\{y\} \\
 & & \quad \text{where } y = \text{GOTO}\{y_0\} + ... + \text{GOTO}\{y_n\} \\
 & & \quad \text{where } y_i := \mathcal{C}[f_i].\text{RET},
\end{array}$$

$$\begin{array}{lll}
(C_{10}) & \mathcal{C}[\text{fold}^L\{g_0,...,g_n\}] & = x := \text{INDUCT}\{y\} \\
 & & \quad \text{where } y = \text{GOTO}\{y_0\} + ... + \text{GOTO}\{y_n\} \\
 & & \quad \text{where } \; y_i \; := \; \text{PR}.\mathcal{C}[map^{E_i}\{P_0, goto\{x\}\}].\text{PAIR} \\
 & & \qquad\qquad\quad . \; \mathcal{C}[g_i] \\
 & & \qquad\qquad\quad . \; \text{RET}, \\
(C_{11}) & \mathcal{C}[\text{map}^L\{h_0,...,h_m\}] & = x := \text{INDUCT}\{y\} \\
 & & \quad \text{where } y = \text{GOTO}\{y_0\} + ... + \text{GOTO}\{y_m\} \\
 & & \quad \text{where } \; y_i \; := \; \mathcal{C}[map^{E_i}\{\mathcal{C}[h_1],...,\mathcal{C}[h_m], goto\{x\}\}] \\
 & & \qquad\qquad\quad . \; \text{CONS}\{i\}.\text{RET},
\end{array}$$

A new combinator *goto* is used to handle recursive calls. The compilation of this combinator is:

$$(C_{12}) \quad \mathcal{C}[goto\{x\}] \quad = \quad \text{GOTO}\{x\}$$

where x is the pointer to the beginning of a sequence of macro code.

Consider the following example of a compilation from the $fold^{\text{list}}$ combinator to macro instructions:

$$\begin{array}{lll}
\mathcal{C}[fold^{\text{list}}\{f_1, f_2\}] & = & \text{INDUCT}\{x\} \\
 & & \quad \text{where } x = \text{GOTO}\{x_1\} + ... + \text{GOTO}\{x_2\}
\end{array}$$

$$\begin{array}{lcll} \text{where} & x_1 & = & \langle \mathcal{C}[\mathit{map}^{E_1}\{p_0, \mathit{goto}\{x\}\}], P_1\rangle.\mathcal{C}[f_1] \\ & & = & \langle \mathcal{C}[\mathit{map}^{1}\{\}], P_1\rangle.\mathcal{C}[f_1] \\ & & = & \langle P_0, P_1\rangle.\mathcal{C}[f_1] \end{array}$$

$$\begin{array}{lcll} \text{where} & x_2 & = & \langle \mathcal{C}[\mathit{map}^{E_2}\{p_0, \mathit{goto}\{x\}\}], P_1\rangle.\mathcal{C}[f_2] \\ & & = & \langle \mathcal{C}[\mathit{map}^{\times}\{p_0, \mathit{goto}\{x\}\}], P_1\rangle.\mathcal{C}[f_2] \\ & & = & \langle \mathrm{MAP}^{\times}\{\mathcal{C}[p_0], \mathcal{C}[\mathit{goto}\{x\}]\}, P_1\rangle.\mathcal{C}[f_2] \\ & & = & \langle \mathrm{MAP}^{\times}\{P_0, \mathcal{C}[\mathit{goto}\{x\}]\}, P_1\rangle.\mathcal{C}[f_2] \\ & & = & \langle \mathrm{MAP}^{\times}\{P_0, \mathrm{GOTO}\{x\}\}, P_1\rangle.\mathcal{C}[f_2] \end{array}$$

The state transitions 13 and 14 show how constructors are stacked on the v stack and how the INDUCT instruction "jumps" to the beginning of some code. In fact, the INDUCT instruction can be thought of as a conditional branch were the constructor in the v stack determines the offset into the table pointed to by the INDUCT instruction.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
13	v	$\mathrm{CONS}\{i\}.c$	d	$\rightarrow$	$u : \mathrm{cons}\{i\}.v$	c	d
14	$v : \langle v_0, \sigma\rangle$	$\mathrm{INDUCT}\{c_i\}_{i=1}^{n}.c$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	c_i	$\mathrm{cont}(c).d$
	$v_0 : \mathrm{cons}_i.v_0'$						

TABLE 7.2. State transition rules for the inductive datatypes

7.4. Linear coinductive datatypes

The record combinator in the previous machine created a record on the value stack with the appropriate closures for each destructor. The linear machine splits this instruction into three separate instructions that build a record with closures on the value stack. The ALLOC instruction allocates n consecutive items in the heap so that the closure instruction will load the ith heap item with the environment (current value stack) and code pair to form a closure for the ith destructor. This consecutive space, with closures, is what the RECORD instruction points to, via a rec node on the value stack. Before each destructor that enters a closure is encountered, a BLDUPDATE instruction pushes the closure the machine is about to enter onto the dump. On exiting the closure, an UPDATE instruction grabs the closure location from the dump and updates it with the computed value and an empty code stream for its code. Subsequent accesses to this closure will immediately load the computed value onto the value stack and exit the closure without updating.

$$
\begin{array}{lllllll}
(C_{13}) & \mathcal{C}[d_i] & = & \multicolumn{4}{l}{\mathrm{DESTR}\{i\}} \\
(C_{14}) & \mathcal{C}[\mathrm{record}^R\{f_0,...,f_n\}] & = & \multicolumn{4}{l}{\mathrm{ALLOC}\{n\}.\mathrm{BLDCLO}\{1,x_1\}. \ldots .\mathrm{BLDCLO}\{n,x_n\}} \\
 & & & \text{where} & x_i & := & \mathrm{BLDUPDATE} \\
 & & & & & . & \mathcal{C}[f_i] \\
 & & & & & . & \mathrm{UPDATE}\{i\} \\
 & & & & & . & \mathrm{RET}, \\
(C_{15}) & \mathcal{C}[\mathrm{unfold}^R\{g_0,...,g_n\}] & = & \multicolumn{4}{l}{x := \mathrm{ALLOC}\{n\}.\mathrm{BLDCLO}\{1,x_1\}. \ldots .\mathrm{BLDCLO}\{n,x_n\}} \\
 & & & \text{where} & x_i & := & \mathrm{BLDUPDATE} \\
 & & & & & . & \mathrm{PR}.\mathcal{C}[g_i].\mathrm{PAIR} \\
 & & & & & . & \mathcal{C}[map^{E_i}\{P_0, goto\{x\}\}] \\
 & & & & & . & \mathrm{UPDATE}\{i\} \\
 & & & & & . & \mathrm{RET}, \\
(C_{16}) & \mathcal{C}[\mathrm{map}^R\{h_0,...,h_m\}] & = & \multicolumn{4}{l}{x := \mathrm{ALLOC}\{n\}.\mathrm{BLDCLO}\{1,x_1\}. \ldots .\mathrm{BLDCLO}\{m,x_m\}} \\
 & & & \text{where} & x_i & := & \mathrm{BLDUPDATE} \\
 & & & & & . & \mathrm{PR}.P_0.\mathrm{DESTR}\{i\}.\mathrm{PAIR} \\
 & & & & & . & \mathcal{C}[map^{E_i}\{h_1,...,h_m, goto\{x\}\}] \\
 & & & & & . & \mathrm{UPDATE}\{i\} \\
 & & & & & . & \mathrm{RET},
\end{array}
$$

The state transitions below show the allocation of heap space for closures, and the instruction which builds a closure in the heap.

Once a closure has been entered and evaluated, it must be updated with the result

	$v \vert h$	c	$d \vert f$	$\rightarrow$	$v' \vert h'$	c'	$d' \vert f'$
15	v	ALLOC$\{n\}.c$	d	$\rightarrow$	$v.\text{rec}\{\text{clo}[i] : (\epsilon, \epsilon)\}_{i=1}^{n}$	c	d
16	$v.\text{rec}\{\text{clo}[i] : (\epsilon, \epsilon)\}_{i=1}^{n}$	BLDCLO$\{i, c_i\}.c$	d	$\rightarrow$	$v.\text{rec}\{\text{clo}[i] : (v, c_i)\}_{i=1}^{n}$	c	d

TABLE 7.3. State transition rules for the building closures

in order avoid recomputation. Transition 17 the pushes the closure on the dump to make it accessible when performing an update to the closure. Transition 18 explicitly updates the value component of a closure to hold the result of the computation and updates the code portion to have an empty code stream. Future references to this closure will return the already computed value.

	$v \vert h$	c	$d \vert f$	$\rightarrow$	$v' \vert h'$	c'	$d' \vert f'$
17	$v : \text{rec}\{v[i] : (v_i, c_i)\}_{i=1}^{n}$	DESTR$\{i\}.c$	d	$\rightarrow$	v_i	c_i	update$(v[i])$.cont$(c).d$
18	v	UPDATE$\{i\}.c$	update$(u).d$	$\rightarrow$	v	c	d
	$u : \text{rec}\{u[i] : (v_i, c_i)\}_{i=1}^{n}$				$u : \text{rec}\{u[i] : (v, \epsilon)\}_{i=1}^{n}$		

TABLE 7.4. State transition rules for updating a closure

7.5. Linear natural numbers

The natural number optimizations of the last chapter also need to be reworked into the linear setting. Like the previous machine, a nat object is used to store a number in the heap eg: succ(succ(zero)) is represented as nat$\{2\}$. Transitions 19 and 20 operate on this natural number object in the heap. The linear natural numbers are divided into the those instructions that deal with the case and those that deal with the fold. Transitions 21 and 22 are the zero and succ cases over nats. The last three transitions deal with folding over a natural number. Loop initialization (the zero phrase) is performed by transition 23. Transition 24 and 25 determine if the machine should continue iteration or exit the loop.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
19	v	$\text{NAT}\{n\}.c$	d	$\rightarrow$	$v' : \text{nat}\{n\}.v$	c	d
20	$\text{nat}\{n\}$	$\text{SNAT}.c$	d	$\rightarrow$	$\text{nat}\{s(n)\}$	c	d
21	$v : \langle v_0, \sigma\rangle$	$\text{CASE\_NAT}\{f_z, f_s\}$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	f_z	d
	$v_0 : \text{nat}\{z\}.v_0'$						
22	$v : \langle v_0, \sigma\rangle$	$\text{CASE\_NAT}\{f_z, f_s\}$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	f_s	d
	$v_0 : \text{nat}\{s(n)\}.v_0''$				$v_0 : \text{nat}\{n\}.v_0''$		
23	$v : \langle v_0, \sigma\rangle$	$\text{LOOPINIT}.c$	d	$\rightarrow$	$v' : \langle v_0', \sigma\rangle$	c	$\text{state}(n, \sigma).d$
	$v_0 : \text{nat}\{n\}.v_0'$						
24	v	$\text{LOOP}(c').c$	$\text{state}(s(n), \sigma).d$	$\rightarrow$	$v' : \langle v, v_2\rangle$	c	$\text{state}(n, \sigma).d$
25	v	$\text{LOOP}(c').c$	$\text{state}(z, \sigma).d$	$\rightarrow$	v	c'	d

TABLE 7.5. State transition for $\mathcal{N}$ datatype

The translations below show how the combinators are compiled to linear natural number instructions.

$$
\begin{array}{llcl}
(C_{17}) & \mathcal{C}[fold^{\mathcal{N}}\{f_z, f_s\}] & = & \text{LOOPINIT}.\mathcal{C}[f_z].a := \text{LOOP}(b) \\
 & & & .\mathcal{C}[f_s].\text{GOTO}\{a\}\ .b := ..., \\
(C_{18}) & \mathcal{C}[case^{\mathcal{N}}\{f_z, f_s\}] & = & x := \text{CASE\_NAT}\{y_0, y_1\} \\
 & & \text{where} & y_0 := \mathcal{C}[f_z].\text{RET}, \\
 & & & y_1 := \mathcal{C}[f_s].\text{RET}
\end{array}
$$

7.6. Linear function (macro) passing

Each function passed as a macro must be individually onto the f stack. A collection of these macros passed to a function form a frame. Transition 26 loads a frame on the f stack while transition 27 invokes a call to a function. When the call to a function is finished, transition 28 unloads the frame from the f stack.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
26	v	$\mathrm{LDPARM}\{\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}\}.c$	d	$\rightarrow$	v	c	d
			f				$\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}.f$
27	v	$\mathrm{CALL}\{c'\}.c$	d	$\rightarrow$	v	c'	$\mathrm{cont}(c).d$
28	v	$\mathrm{UNLOAD}.c$	d	$\rightarrow$	v	c	d
			$\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}.f$				f

TABLE 7.6. State transition for linear function passing

The compilation of the *call* combinator from the original machine is:

$$(C_{19}) \quad \mathcal{C}[call\{c', (c_1, ..., c_n)\}] \quad = \quad \mathrm{LDPARM}_{i=1}^{n}.\mathrm{CALL}\{c'\}.\mathrm{UNLOAD}$$

When the called function reaches a macro, it selects the code from the current (top) frame of the f stack with transition 29. When a macro is invoked, the machine must push the current f frame onto the dump, effectively restoring the f stack to the environment the macro must operate in. After the macro has completed computation, transition 30 returns from the call and restore the f stack to its former state. This is accomplished by popping the frame of the dump and pushing it onto the f stack.

	$v\|h$	c	$d\|f$	$\rightarrow$	$v'\|h'$	c'	$d'\|f'$
29	v	$\mathrm{PARM}(j).c$	d	$\rightarrow$	v	c_j	$\mathrm{reload}(\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}).d$
			$\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}.f$				f
30	v	$\mathrm{RELOAD}.c$	$\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}.d$	$\rightarrow$	v	c	d
			f				$\mathrm{reload}(\mathrm{parm}\{p[i] : c_i\}_{i=1}^{n}).f$

TABLE 7.7. State transition for linear function passing

The compilation of the *sel* combinator from the original machine is:

$$((C_{20})) \quad \mathcal{C}[sel\{i\} \quad = \quad \mathrm{PARM}\{i\}.\mathrm{RELOAD}$$

Chapter 8
Results and Conclusions

This thesis shows how the Charity Abstract Machine can be successively refined to levels closely approximating a physical machine. The third machine described (the linear abstract machine) is roughly 20 times faster than the original (first) machine, which was programmed in SML. Furthermore, as the new machine implements sharing, on some programs the gain one can expect is considerably greater than this.

Interestingly, when the third machine was first implemented, it was about 5% to 10% slower than the second machine. Since the overhead of interpreting each instruction is constant, but size of the code generated in the linear machine is larger, the overall interpreting cost becomes a more significant factor in the execution time. The decrease in speed is, thus, an indication that the cost of interpreting instructions in the third machine has outweighed the gains from better approximating the physical machine. Thus, a threshold was reached in attaining speed gain through this refinement process.

Despite this, the linear machine has been used in preference to the second machine because, not only is it more amenable to "peephole" optimizations, but it would also be easier to compile down to actual machine instructions (thus removing the interpreting overhead).

Implementing the Charity Abstract Machine in the C language yielded not only a faster, but also a more portable implementation. However, at a cost: the development time was considerably longer than that required for the original SML version.

The contributions of this thesis are:

- Completion and correction of the design of the basic Charity Abstract Machine initiated by M. Hermann (CHARM) [Her92].
- Specification of the macro passing.
- Design of the linear abstract machine.
- Development of compilation routines into the instructions for these machines.
- The reimplementation of CHARM in C.

Possible future work with the Charity abstract machine could involve:

- Optimizing the translation from term logic to combinators. Improving this translation would reduce the generation of unnecessary combinators. Specifically, the abstraction, inductive and coinductive terms should check to see if there are any free variables within their body before generating the code to create the environment;
- Compiling code directly to native machine code to eliminate the cost associated with interpreting in the "byte–code". Instead of interpreting each instruction, actual machine instructions could be generated and executed;
- Providing direct access to variables rather that looking up values by projecting the environment;
- Adding built–in types such as integers and strings, along with primitive operations that act on these types.
- Proving the correctness the machines and their compilation procedures.
- Optimizing combinator to macro instruction compilation (for example, peephole optimization).

Bibliography

[ASU85] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, Reading, Massachusetts, 1985.

[Bac78] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BW88] Richard Bird and Phillip Wadler. *Introduction to Functional Programming*. Series in Computer Science (C. A. R. Hoare ed.). Prentice-Hall, London, 1988.

[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 Archite cture*, 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.

[Coc] R. Cockett. Lecture notes in applied mathematics: Seminar on category theory.

[CS95] R. Cockett and D. Spencer. Strong categorical datatypes ii: A term logic for categorical programming. *tcs*, 139:69–113, 1995.

[Cur86] P-L Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* John Wiley & Sons, Inc, New York, 1986.

[FH88] A. J. Field and P. G. Harrison. *Functional programming.* Addison–Wesley, New York, 1988.

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

[FW87] Jon Fairbairn and Stuart C. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, OR, September 1987. Springer-Verlag.

[Gor88] M. J. C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

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

[Hen80] P. Henderson. *Functional Programming – Applications and Implementation.* Prentice–Hall, London, 1980.

[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.

[Hug84] R. Hughes. *The design and implementation of programming languages.* PhD thesis, University of Oxford, 1984.

[Hug89] R. J. M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[Jon] S. Peyton Jones. Functional languages implementation tutorial.

[Jon92] S. L. Peyton Jones. “Implementing lazy functional languages on stock h ardware: 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.

[Kie85] Richard B. Kieburtz. The G-machine: A fast, graph-reduction evaluator. In *Conference on Functional Programming Language and Computer Architecture*, pages 400–413, Nancy, France, 1985. IFIP.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Pau91] L. Paulson. *Functional Programming – Applications and Implementation.* Cambridge University Press, Cambridge, 1991.

[Set89] R. Sethi. *Programming Languages Concepts and Constructs.* Addison–Wesley, Murry Hill, New Jersey, 1989.

[Spe93] D. Spencer. *Categorical Programming with Functorial Strength.* PhD thesis, Oregon Graduate Institute, 1993.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. In *Software — Practice and Experience*, volume 9, pages 31–49. John Wiley and Sons, September 1979.

[Tur85] D. Turner. “Miranda – a non–strict functional language with polymorphic types”. In G. goos and J. Hartmanis, editors, *Functional Programming Languages and Computer Archite cture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer–Verlag, September 1985.

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

[Yee] B. Yee. The charm project: A back end to the charity interpreter.

Appendix A
Formal Definition of Charity

The formal definition of Charity originally appeared in [CF92].

A.1. Variable bases

For each type we have a set of $\{x, y, z, ...\}$ variables (in fact, the type is inferred). A **variable base** is then defined follows:

- () is a variable base of type 1,
- If x is a variable, then x is a variable base with type $type(x)$,
- If v_0 and v_1 are variable bases *with no variables in common*, then (v_0, v_1) is a variable base where

$$type((v_0, v_1)) = type(v_0) \times type(v_1).$$

A.2. Terms

A **term** is defined as follows:

- () is a term of type 1,
- If t is a term where $type(t) = X \times Y$, then $p_0(t)$ and $p_1(t)$ are terms, where $type(p_0(t)) = X$ and $type(p_1(t)) = Y$,
- If t_0 and t_1 are terms, then (t_0, t_1) is a term, where $type((t_0, t_1)) = type(t_0) \times type(t_1)$,

- If w is a variable base, and t is a term, where $type(w) = type(t)$, then

$$\{w \mapsto t'\}(t)$$

  is a term of type $type(t')$, and the variables in w are bound in t'.

- If $t_1, ..., t_n$ are terms where $type(t_i) = E_i(A, L(A))$ then $c_1(t_1), ..., c_n(t_n)$ are terms where $type(c_i(t_i)) = L(A)$.

- If t is a term where $type(t) = L(A)$ and $v_1, ..., v_n$ are variable bases where $type(v_i) = E_i(A, L(A))$ and $t_1, ..., t_n$ are terms where $type(t_1) = ... = type(t_n) = B$ then

$$\left\{ \begin{array}{ccc} c_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ c_n(v_n) & \mapsto & t_n \end{array} \right\}(t)$$

  is a term (the case expression) of type B. The variables in $v_1, ..., v_n$ are bound in $t_1, ..., t_n$ respectively.

- If t is a term where $type(t) = L(A)$ and $v_1, ..., v_n$ are variable bases where $type(v_i) = E_i(A, X)$ and $t_1, ..., t_n$ are terms where $type(t_1) = ... = type(t_n) = X$ then

$$\left\{\!\!\left| \begin{array}{ccc} c_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ c_n : v_n & \mapsto & t_n \end{array} \right|\!\!\right\}(t)$$

  is a term (the fold) of type X. The variables in $v_1, ..., v_n$ are bound in $t_1, ..., t_n$ respectively.

- If t is a term where $type(t) = L(A_1, ..., A_m)$ and $v_1, ..., v_m$ are variable bases where $type(v_j) = A_j$ and $t_1, ..., t_m$ are terms where $type(t_j) = B_j$ then

$$L\left\{ \begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array} \right\}(t)$$

is a term (the map expression) of type $L(B_1, ..., B_m)$. The variables in $v_1, ..., v_m$ are bound in $t_1, ..., t_m$ respectively.

- If t is a term where $type(t) = S$ and v is a variable base where $type(v) = S$ and $t_1, ..., t_n$ are terms where $type(t_j) = F_j(A, S)$ then

$$\left(\!\left| v \mapsto \begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right|\!\right)(t)$$

  is a term (the unfold) of type $R(A)$. The variables in v are bound in $t_1, ..., t_n$ respectively.

- If $t_1, ..., t_n$ are terms where $type(t_j) = F_j(A, R(A))$ then

$$\left(\begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right)$$

  is a term (the record) of type $R(A)$.

A.3. Abstracted maps

A program is not a term, but an **abstracted map**. This is a pair

$$\{v \mapsto t\}$$

where v is a variable base containing all the free variables of the term t.

Appendix B
A sample Charity program

B.1. Quicksort

The quicksort program for sorting elements can be defined in Charity. Recall that the quicksort algorithm first splits a list into a tree, then performs an inorder traversal of to tree to collect the result.

The data definition required to hold the values in a tree is the `coBTree` data type.

```
data C -> coBTree(A) = deBtree: C -> sf(A * (C * C)).
```

Leaves are labeled with the `ff` constructor, while nodes containing values along with their left and right children are labeled with `ss`.

The first function required is `split` which takes a list and separates it into two lists, based on some predicate `P`.

```
def split {P} (a, L) =
   {| nil: ()            => ([],[])
    | cons: (b, (L1, L2)) => { true()  => (cons(b, L1), L2)
                            | false() => (L1, cons(b, L2))
                            }(P(b,a))
    |}(L).
```

From `split`, the function `pivot` will separate the pivot from those elements “less than” the pivot, and “greater than” the pivot.

```
def pivot {P} (L) =
   { nil()       => ff()
   | cons(a, L)  => ss(a, split{P} (a, L))
   }(L).
```

With unfold over the original list, a `coBtree` is created where the root of the tree is the first element in the list.

```
def pivots{P}(L) = (| x => deBTree: pivot{P}(x) |)(L).
```

To collect the values in to the nodes of the cotree, an inorder traversal the coBTree is required. The function `collect` performs the traversal by taking the tree apart from the root down, until a leaf is encountered. The right side of the cotree and the current node, starting from the root node, is stacked onto a dump so that the left side to the coBtree can be processed first. When a leaf on the left on the left side of the tree is encountered, the value of the node is appended to the result and the ride side of the node is popped off the dump using the `pop` function. The collect function continues collecting values down the right side of the tree, until all nodes have been collected.

```
def pop(acc,dump) = { nil()             => (acc,((deBTree:ff()),[]))
                    | cons((a,t),dump') => (cons(a,acc),(t,dump'))
                    }(dump).

def collect(tree, bnd) =
 {| zero:()                           => ([],(tree,[]))
  | succ:(acc,((deBTree:ff()), dump)) => pop(acc,dump)
        |(acc,((deBTree:ss(a,(tl,tr)) => (acc,(tl,cons((a,tr),dump)))
  |}(bnd).
```

Finally the quicksort function first generates the coBtree, then performs a traversal of the tree, collecting all of the values into a list. Notice that the upper bound of the traversal is $2 \times n + 1$, since each node must be visited twice during the inorder traversal.

```
def quick_sort{P}(L) =
   {len => p0(collect(pivots{P}(L),succ(add(len,len))} (length(L)).
```