THE UNIVERSITY OF CALGARY

An Implementation of Charity

by

Min Zeng

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

March, 2003

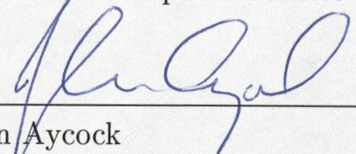
© Min Zeng 2003

THE UNIVERSITY OF CALGARY

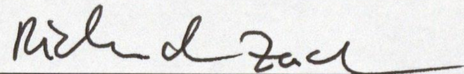
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "An Implementation of Charity" submitted by Min Zeng in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. Robin Cockett
Department of Computer Science



Dr. John Aycock
Department of Computer Science



Dr. Richard Zach
Department of Philosophy

March 26, 2003

Date

Abstract

This thesis describes an implementation of core-Charity which consists of a virtual machine (VMC) and a separate compiler to translate Charity code into code for that machine.

The design of the virtual machine was inspired by the ABC machine of Clean. It is a general purpose byte code machine which incorporates garbage collection and the separation of data into basic types and heap pointers.

The core-Charity compiler concentrates on providing an efficient implementation of core Charity code into the virtual machine. Previous implementations suffered from excessive heap usage which necessitated frequent garbage collection. In this implementation a variety of techniques are used to reduce heap usage and improve execution speed.

The resulting Charity programs are substantially faster than those produced by previous Charity implementations.

Acknowledgements

I would like to express my sincere gratitude to Dr. Robin Cockett for guiding me through the process of researching and completing this thesis. This work would not have been possible without his support, encouragement and patience.

I am also grateful to Jeff Green, Dana Harrington and David Pereira, for reading the draft of this thesis, correcting numerous English errors in it and giving me useful suggestions.

Finally, thanks to my wife, Winnie, for her understanding, patience and love.

Table of Contents

Approval Page ii

Abstract iv

Acknowledgements v

Table of Contents vi

1 Introduction 1
- 1.1 The Charity Programming Language . . . 1
  - 1.1.1 Overview . . . 1
  - 1.1.2 Inductive datatypes . . . 2
  - 1.1.3 Coinductive datatypes . . . 5
  - 1.1.4 Combinators and macros . . . 6
  - 1.1.5 Higher-order functions . . . 7
- 1.2 Implementations of Charity . . . 8

2 The Virtual Machine for Charity (VMC) 11
- 2.1 Introduction . . . 11
- 2.2 Overview . . . 12
- 2.3 The VMC assembly language . . . 15
- 2.4 Other virtual machines . . . 18
  - 2.4.1 The Charity Abstract Machine . . . 18
  - 2.4.2 The Spineless Tagless G-Machine . . . 20
  - 2.4.3 The abstract machine for Standard ML of New Jersey . . . 22
  - 2.4.4 The ABC machine . . . 24
- 2.5 Representing Charity data in VMC . . . 26
  - 2.5.1 Primitive data and tuples . . . 26
  - 2.5.2 Inductive Data . . . 27
  - 2.5.3 Coinductive data representation . . . 29

3 Intermediate representation for Charity code 34
- 3.1 Structure of Intermediate Representation . . . 35
  - 3.1.1 Representation of type expressions and multi-functions . . . 35
  - 3.1.2 Symbol Table . . . 37
- 3.2 Fold, Unfold and Map . . . 41
- 3.3 Transforming combinator invocations . . . 48

3.3.1 Introduction 48
3.3.2 Environment variables 51
3.3.3 Generating new functions 53
3.3.4 Avoiding code explosion 56
3.4 Simplification and Optimization 60
3.5 Polymorphic functions and basic values 62
3.6 Type Checking Again 65

4 VMC Code Generation **72**
4.1 Overview 72
4.2 Code Generation Rules 73
4.3 Optimizations 81
4.3.1 Simple inductive datatypes 82
4.3.2 Inductive datatype with only one constructor 83
4.3.3 Higher-order coinductive datatypes with only one destructor . 84
4.3.4 Peephole optimizations on VMC code 85

5 Results and Conclusion **88**

Bibliography **92**

A VMC specification **97**
A.1 Formal Syntax of VMC program 97
A.2 Data 98
A.2.1 Basic data 98
A.2.2 Pointer data 98
A.3 Instruction Set 98
A.3.1 Arithmetic instructions 100
A.3.2 Logical and bit instructions 102
A.3.3 Comparison instructions 102
A.3.4 Data conversion instructions 104
A.3.5 Branch instructions 105
A.3.6 Constant instructions 108
A.3.7 Stack instructions 108
A.3.8 Tuple instructions 110
A.3.9 Array instructions 111
A.3.10 String instructions 113

B VMC interpreter user's manual **114**
B.1 Configuration 114
B.2 Command line syntax 115
B.3 Commands 116

List of Figures

2.1 Overall structure of VMC 13
2.2 How a tuple is stored in the heap 14
2.3 How `succ zero` is stored in the heap 28
2.4 How `(fst:3, snd:5)` is stored in the heap 32

3.1 cons1 vs. cons2 38
3.2 Two possible storage formats of [5,6] 64

Chapter 1

Introduction

1.1 The Charity Programming Language

1.1.1 Overview

Charity [10, 11] is a unique programming language motivated by a category theoretic model of computation [30]. In typical functional style [8, 7], a Charity program is a collection of functions. Each function generates its output solely from its input and no side-effects are allowed, thus there are no updateable variables or assignment statements. Charity programs are strongly typed, that is, the types of functions are resolved at compile time and type mismatch errors can never arise at run time. However, some fundamental differences exist between Charity and other functional languages:

- General recursion, which plays a very important role in functional languages, is not allowed in Charity. The sacrifice of general recursion gives Charity a special property: all Charity programs are guaranteed to terminate. This property is very useful in program transformation and verification.
- Charity is a datatype-oriented language, its power comes from defining new datatypes and the transformations between them. In addition to the inductive datatypes available in other functional languages, Charity has another class of datatypes called coinductive datatypes, which arise as the categorical dual

of inductive types. Coinductive datatypes can represent potentially infinite data structures, model *states* in imperative languages, and simulate *objects* in object-oriented languages. Coinductive datatypes greatly enhance the computation power of Charity.

- Charity can be both lazy and strict. Since Charity programs always terminate, whether or not arguments to functions evaluated lazily or strictly will not change the semantics of Charity programs. Coinductive data in Charity is always evaluated lazily. It is possible, therefore, to implement Charity as a strict language and use coinductive data when laziness is needed.

In the following sections, we will use simple examples to show the primary facilities in Charity; the detailed syntax and semantics of these facilities will be discussed in other chapters.

1.1.2 Inductive datatypes

Inductive datatypes are similar to datatypes in other functional languages. Below are the datatype definitions for boolean and natural numbers:

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

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

The keyword `data` starts a datatype definition. The expression `bool -> C` specifies the name of the datatype: `bool`, and the *state variable*: `C`. The *state variable* can be viewed as an alias for the datatype being defined. `false`, `true`, `zero` and

`succ` are *constructors*, special functions that *construct* inductive data from data of other types. The expression following a constructor is the type of the constructor. The type of `false` is `1 -> C`, which means that its input type is `1` and its output type is `C`, which is the alias for `bool`. `1` stands for the *unit* datatype; there is only one member of this type and it is represented by (). The output type of a constructor must always be the *state variable*. The type of `succ` is `C -> C`, which says that `succ` takes data of type `nat` as input and generates data of type `nat` as output. For example, `succ zero`, the representation of the natural number "1", is of type `nat`, .

Inductive datatypes can also have one or more *type variables*. A datatype with type variables is called a *polymorphic datatype*. For example, we use the following definition for the list datatype:

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

Here, `A` is a *type variable* whose value can be any datatype. For example, `list(nat)` is the type of a list of natural numbers, `list(bool)` is the type of a list of boolean values, and `list(list(bool))` is the type of a list whose elements are lists of boolean values. The input type of `cons` is `A,C`, which says that `cons` has two input parameters: the first parameter is of type `A`, and the second argument is of type `C`, i.e. `list(A)`. The concrete type that `A` represents can often be inferred from the arguments of `cons`. For example, from `cons(true,nil)` we can infer that `A` equals type `bool`, so the type of the expression is `list(bool)`.

Three operations are by default available with a defined inductive datatype: *case*, *fold* and *map*. Below are three examples in that order:

```
def not = x =>{ false => true
              | true  => false
              } x.

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

def addone = L => list{succ}L.
```

The above examples define three functions: `not`, `add` and `addone`. The `not` function illustrates the use of *case*. The part inside the {} is a *case*. In general, a *case* operation takes an inductive value as input and examines the "root" of the value to determine which action to perform. In this example, it checks the constructor of the boolean variable `x` and returns the opposite value of `x`.

The add function illustrates the use of *fold*. The part inside the `{|  |}` is a *fold*. A *fold* takes an inductive value as input, examines the root of the value, applies the *fold* itself over the recursive part of the value, and then performs the action specified in the *fold*. In this example, the *fold* examines `x`: if it is `zero`, `y` is returned; if it is `succ`, the fold itself is applied to the argument of `succ` (which is also a natural number) to get a natural number `n`, thereupon returning `succ n`.

The addone function illustrates the use of *map*. Here `list{succ}` is the *map*. The input type of a *map* is a *polymorphic* datatype that has one or more *type variables*. A *map* may take one or more functions, with each function corresponding to a type variable in the polymorphic datatype. It applies the functions to the type variable part of the data while preserving the structure of the data. In this example, *map* applies `succ` to every natural number in the list `L`.

1.1.3 Coinductive datatypes

One way in which Charity differs from other functional languages is that Charity allows coinductive data. Coinductive datatypes provide a means of representing potentially infinite structures which are evaluated lazily. Here is the definition of infinite lists as a coinductive datatype:

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

Note that the *state variable*, `C`, is now at the left side of the arrow. This distinguishes a coinductive datatype definition from an inductive datatype definition. `head` and `tail` are *destructors*, special functions that *destruct* coinductive data into the data of other types. Coinductive datatypes are similar to abstract datatypes: the coinductive datatype definition defines the 'interface' of the data.

To create coinductive data, three operations are available: *record*, *unfold* and *map*. Below is an example:

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

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

def newnats  = () => inflist{succ}(nats).
```

The first function, `nats`, generates an object which represents an infinite list of natural numbers $1, 2, 3, \ldots$, by using an *unfold*. `x` represents the *internal state* of the object, which is a natural number. The expression following `head:` and `tail:` specifies the action to take when the corresponding destructor is called. When `head`

is called, the internal state `x` is returned, which serves as the head of the infinite list. When `tail` is called, the internal state is incremented by 1, then *the unfold itself is recursively applied on the new internal state to generate a new object*, which will be returned as the result. `succ zero` is the initial internal state of the unfold, so the infinite list starts from 1.

The function `zeronats` generates the infinite list `0,1,2,...`, by using a *record*. A record works by simply giving the values to be returned by each destructor. So when `head` is applied on the object, `zero` is returned, when `tail` is applied on the object, the infinite list `1,2,...` is returned lazily.

The last function `newnats` uses *map* to convert `1,2,...` into `2,3,...`. Conceptually, the *map* for coinductive data is very similar to the *map* for inductive data, since the type variable part of the data is transformed but the structure of the datatype remains the same. *The difference is that the map on coinductive data is always evaluated lazily.*

1.1.4 Combinators and macros

The original Charity was a first order language and did not have any higher-order functions. However, Charity provides a simple facility to pass functions as parameters. Here is an example:

```
def filter{f} =
    L => {|  nil: ()   => nil
          | cons: x,l => { true  => cons(x,l)
                          | false => l
                          }(f x)
          |} L.
```

Quite often we want to filter out elements in a list that do not satisfy a certain condition. For generality, the condition is passed to `filter` as a functional parameter `f`. The functional parameter `f` is called a *macro*, and the function `filter` is called a *combinator*.

To filter out `zeros` in a list of natural numbers, we only need to define a `nonzero` predicate and then apply `filter{nonzero}` over the list.

```
def nonzero = x => { zero     => false
                   | succ(x) => true
                   } x.
```

A combinator name along with its macro arguments, e.g. `filter{nonzero}`, is called a *combinator invocation*.

1.1.5 Higher-order functions

Although functions can be passed as arguments to combinators, functions were not first-class citizens in the original Charity. A function could not be returned as the result of another function or be stored in a data structure. Higher-order capabilities were introduced by allowing the destructors to have extra parameters [28]. This enhancement effectively gives Charity the capability of higher-order functions, since, to introduce higher-order functions, all that is required is to define the following datatype:

```
data C -> exp(A,B) = fn: A,C ->B.
```

`fn` is a destructor with an additional parameters `A`. A *record* of the above type looks like `(fn:x=>...)`, that is, the value specified for `fn` is a function. Since coinductive data can be passed as parameters and returned as results, we achieve the power

of a higher-order language by using coinductive datatypes to represent functions. Below is the `filter` example rewritten using coinductive data to pass functions as arguments:

```
def filter =
    f, L => {|  nil: ()   => nil
             | cons: x,l => { true  => cons(x,l)
                             | false => l
                             }(fn(x,f))
             |} L.

def nonzero = (fn: x => { zero    => false
                        | succ(x) => true
                        } x
              ).

def filterzero = L => filter(nonzero, L).
```

1.2 Implementations of Charity

The original implementation, developed by Tom Fukushima, was an interpreter written in SML, which was an implementation of core-Charity (i.e. without pattern-matching) which ran on a very basic categorical machine. The next implementation [36] was an interpreter written in C, and was significantly faster (over 100 times) than the original SML implementation but also took a significantly longer time to develop. Though reasonably fast, this C implementation suffers from the following problems:

- The front-end and back-end of the interpreter are very closely integrated, which made modification and debugging very difficult.

- Writing the front-end in C is complicated and error prone, because C is not good at handling complex data structures and does not provide any automatic memory management facility. Adding new features to Charity meant exploring tens of thousands of lines of C source code. Some bugs were still there after years of usage.

- The back-end is based on CHARM (Charity Abstract Machine) [17, 36], a simple machine which, however, suffers from excessive heap usage. It is also hard to add new features to Charity without significant changes to CHARM.

This thesis describes a different approach, which has the following advantages:

- The front-end and back-end are two separated programs. The front-end is a parser that translates core-Charity code into a clearly defined, platform-independent low-level language. The back-end is an interpreter for this low-level language. Modifications of one will not affect the other as long as the low-level language remains unchanged.

- As compilation speed is not a major issue, the front-end is written in the functional language SML, a more suitable tool than C for writing a compiler. The SML code is considerably shorter and clearer, thus is much easier to maintain.

- The back-end is based on a virtual machine called VMC (Virtual Machine for Charity), which is a simple, flexible and efficient virtual machine. The VMC is designed to model a concrete machine instead of specific language features, so that new features of Charity can be implemented efficiently without changes to

the target VMC. It is also potentially straightforward to translate VMC code into native machine code.

The next chapter will describe the VMC, and the following chapters will describe the translation from Charity to VMC. A complete specification of VMC is in appendix A.

Chapter 2

The Virtual Machine for Charity (VMC)

2.1 Introduction

The Virtual Machine for Charity (VMC) is an abstract machine with a stack-based architecture and instruction set. The instruction set of the VMC is used as target language in the translation of Charity, that is, Charity code is compiled into VMC code, and then executed by an interpreter or further translated into concrete machine code. The advantages of this approach are:

- A more structured implementation can be achieved. By using a virtual machine, we have a clearer view of how the language is conceptually implemented and what the trade-offs are.
- Implementations can be ported more easily to different platforms because we only need to write a VMC interpreter/compiler for each platform, while the compiler that translates Charity code into VMC code remains intact.
- By separating the front-end and back-end into different programs, we can optimize each according to criteria relevant to its function. The front-end can be written in a functional language, such as SML, making it easier to understand and modify, while the back-end (interpreter) can be implemented in C/C++ for speed.

Consequently, the VMC is designed with the following objectives in mind:

Portability The VMC should not assume any particular processor details, such as addressing modes, number of registers, etc.

Simplicity The VMC should be simple enough to be implemented easily and efficiently on many different platforms.

Efficiency The VMC instruction set should be able to interpret Charity code efficiently. Overhead introduced by this translation should be minimized.

2.2 Overview

Figure 2.1 illustrates the overall structure of VMC, which consists the following components:

B-stack This is the stack for basic values, such as integers, real numbers, code addresses, etc. Each unit on B-stack is 32 bits. Data smaller than 32 bits, such as characters, will be expanded into 32 bits before being put on B-stack. 64 bits basic data, such as long integers, will occupy 2 units. Units on B-stack are referenced by their relative position from top of the stack, with the top most unit being unit 0.

P-stack Pointers to heap elements are placed in this stack. As for the B-stack, every pointer on the P-stack is 32 bits and is referenced by its relative position from top of the stack.

Heap The heap is used to contain compound data structures. Unlike other virtual machines, such as the ABC machine [26], which uses a special node format to

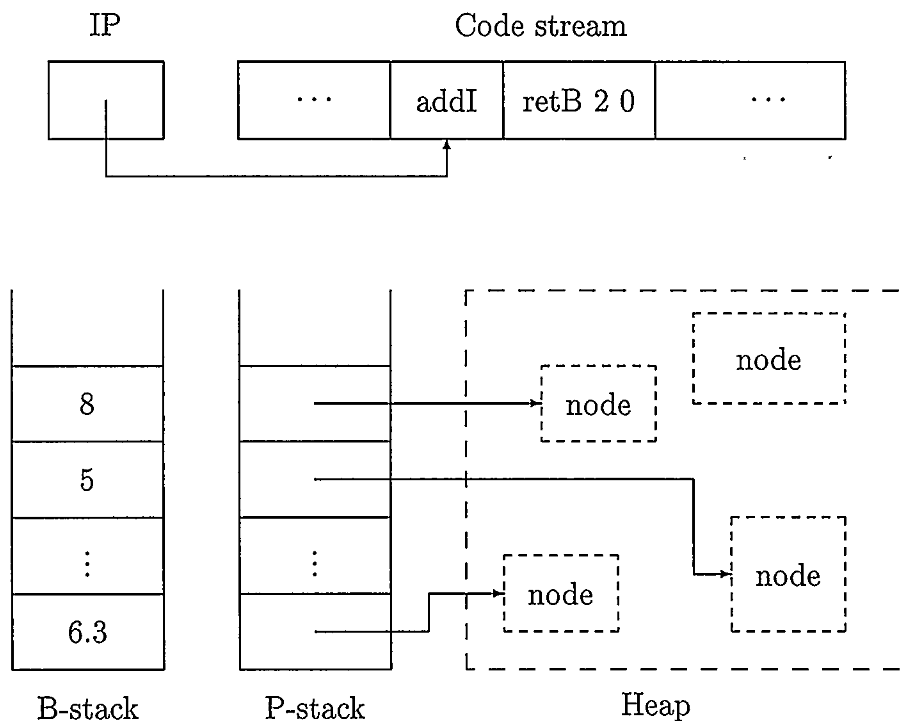


Figure 2.1: Overall structure of VMC

store inductive data, the VMC is not defined with respect to inductive or coinductive datatypes, everything in the heap is either a tuple or an array. Inductive data is stored as a tuple with the first element indicating the constructor. Coinductive data is stored as a tuple of closures, one for each destructor, where each closure is a tuple composed of a code entry and its arguments. A tuple can contain an arbitrary number of basic values and pointer values.

Figure 2.2 shows how a tuple composed of m basic values and n pointer values is stored in the heap (each row is 32 bits wide). The first row is the header of the tuple that indicates the number of basic values and pointer values in the tuple.

m	n
basic value 1	
...	
basic value m	
pointer 1	
...	
pointer n	

Figure 2.2: How a tuple is stored in the heap

The format of the header is not accessible by VMC code and is internal to the virtual machine, allowing machine-specific implementation while retaining portability. This layout of tuples makes it easy to tell pointers from basic values, which is important for garbage collection.

Garbage collection [35, 5] in the VMC is implicit, and is automatically invoked whenever a program tries to allocate memory from the heap but is unable to do so due to lack of space. The current implementation used a simple 2-space-copying collector [14], but more advanced garbage collectors, such as generational garbage collectors [24, 4, 27], could be used to improve performance.

Code stream The VMC code sequence to be executed.

IP Instruction Pointer, a register that points to the next instruction to be executed.

2.3 The VMC assembly language

The VMC assembly is best introduced by an example such as the following, which sums 1...100:

```
_start:                 ; program starts here
    constI 1            ; push parameter 1 on B-stack
    constI 100          ; push parameter 100 on B-stack
    call _sum_m2n       ; call subroutine _sum_m2n
    halt                ; stop
_sum_m2n:               ; sum takes two parameters, m and n
    dupB 1              ; push m
    dupB 1              ; push n
    gtI                 ; m>n?
    ifzero  @L          ; no, continue
    constI 0            ; yes, put 0 on B-stack as result
    retB 2 0            ; return
@L:
    dupB 1              ; push m
    constI 1            ; push 1
    addI                ; m+1
    dupB 1              ; push n
    call _sum_m2n       ; recursion, sum_m2n(m+1,n)
    dupB 2              ; push m
    addI                ; m+sum(m+1,n)
    retB 2 0
```

A VMC program is composed of labels, comments and instructions. In the above example, `_start`, `_sum_m2n` and `@L` are labels, which indicate positions in the program. Labels always start with either a `_`, indicating a global label, or a `@`, indicating a local label. Allowing for local labels helps to avoid name clashes between source files. Text from a semicolon to the end of line is treated as a comment and is ignored by the VMC translator. Instructions always start with an *operator* followed by a list of *operands*, separated by spaces. A formal specification of the instruction set can be

found in appendix A. Here we describe some of the core instructions.

constI i: push the constant integer i onto the B-stack. For example, if the content of B-stack is $5.6.\bot$, with 5 being the top item, then after executing `constI 1`, the contents of the B-stack becomes $1.5.6.\bot$.

dupB i: push the ith item on the B-stack onto the B-stack. For example, if the contents of the B-stack is $3.5.7.\bot$, then after executing `dupB 1`, it becomes $5.3.5.7.\bot$.

dupP i: push the ith item on the P-stack onto the P-stack. Similar to `dupB`.

moveB i j: change the jth item on the B-stack to the value of the ith item on the B-stack. For example, if the content of the B-stack is $3.5.7.\bot$, then after executing `moveB 0 2`, the contents of the B-stack becomes $3.5.3.\bot$.

moveP i j: change the jth item on the P-stack to the value of the ith item on the P-stack. Similar to `moveB`.

newtuple m n: Create a tuple in heap, which consists of the top m items on the B-stack and the top n items on the P-stack. The top m items on the B-stack and the top n items on the P-stack are popped off and a pointer to the new tuple is pushed onto the P-stack. For example, if the contents of the B-stack is $5.6.7.\bot$, the contents of the P-stack is $p.q.\bot$, then after executing `newtuple 2 1`, the contents of the B-stack becomes $7.\bot$ and the contents of the P-stack becomes $p'.q.\bot$, where p' points to the tuple $(5.6, p)$ in the heap.

detuple Pops off the top item on the P-stack, which points to a tuple in the heap, then pushes all the basic values in the tuple onto the B-stack and pushes all

the pointers in the tuple onto the P-stack. This instruction can be viewed as the reverse operation of `newtuple`. For example, if the contents of the B-stack is $7.\bot$, the contents of the P-stack is $p'.q.\bot$, where p' points to a tuple $(5.6, p)$, then after executing `detuple`, the contents of the B-stack becomes $5.6.7.\bot$, and the contents of the P-stack becomes $p.q.\bot$. Note that a `newtuple` instruction followed immediately by a `detuple` instruction has no effect on the virtual machine except that it creates a node in the heap which will be garbage collected. An optimization described in section 4.3.4 is based on this observation.

call L**:** Pushes the current value of IP (instruction pointer) onto the B-stack and then changes IP to the address of L. This instruction is usually used to call a subroutine.

xcall: Exchange the values of IP with the top value on the B-stack. This instruction is used to call a subroutine whose address is at the top of the B-stack.

retB m n**:** This instruction is used to return from a subroutine which has m basic arguments and n pointer arguments and returns a basic value. The topmost item on the B-stack will be the result of the subroutine. The item right below the result is the return address. The top $m + 2$ items on the B-stack and the top n items on the P-stack are popped off, then the result is pushed back onto the B-stack, and the IP is changed to the return address. For example, if the contents of the B-stack is $10.L.4.6.2\bot$, the contents of the P-stack is $p.q.r.\bot$, then after executing `retB 2 1`, the contents of the B-stack becomes $10.2.\bot$, the contents of the P-stack becomes $q.r.\bot$, and the IP points to L.

retP m n**:** This instruction is similar to `retB`, except that the result is now at the top of the P-stack and the return address is on top of the B-stack. So $m+1$ items are popped off the B-stack, $n+1$ items are popped off the P-stack, then the result is pushed back onto the P-stack, and IP is set to the return address.

case $L_0 \cdots L_n$**:** This instruction pops off the top item from the B-stack, which is a small non-negative integer i, and sets IP to L_i. For example, if the contents of B-stack is 1.3.$\perp$, then after executing `case @one @two @three`, the contents of B-stack becomes 3.$\perp$, and IP points to `@two` (Note that the first label corresponds to 0) .

2.4 Other virtual machines

Using a virtual machine to implement a functional language is quite common. Here we review four other virtual machines and compare them to the VMC.

2.4.1 The Charity Abstract Machine

The previous implementation of Charity is based on the Charity Abstract Machine (CHARM) [17, 36], which is an adaptation of the Categorical Abstract Machine [12]. Charity programs are expressed as typed combinators and evaluated through a set of rewrite rules. The CHARM consists of:

- A value stack holding current value of a computation. Actually it is not a real stack, as at any time, it holds just one value: a pointer to a heap item. It would be more accurate to call it the current value register.

- A code stack listing the remaining sequence of instructions to execute, similar to the Code Stream plus IP in the VMC. Actually code is not pushed or popped, only the code pointer is manipulated.
- A dump stack holding continuations. A continuation will be pushed onto the dump stack when calling a function or evaluating a pair.
- A function stack holding a list of functions passed as parameters.
- A heap to store products, inductive and coinductive data.

The number of instructions in CHARM is small, and the instructions closely resemble the operations in category theory. So the CHARM is relatively easy to implement, and the translation from Charity to CHARM is not complicated. However, CHARM suffers from excessive heap usage for the following reasons:

- The environment of a function is passed to the function as a pair in the heap with the environment as the second element of the pair. Furthermore, if there are multiple environment variables, they have to be organized as nested pairs.
- As the value stack is not a real stack, multiple arguments to a function must be stored in a heap node and passed as a single argument.
- There are no n-tuples in CHARM. Rather, n-tuples in Charity are translated into nested pairs which take up much more space than n-tuples.
- Basic data is boxed.

2.4.2 The Spineless Tagless G-Machine

The Spineless Tagless G-machine [22] is an abstract machine designed to support lazy higher-order functional languages. It is a synthesis of the G-machine [19] and the TIM machine [13], borrowing the best of both architectures. The STG machine is the backend for Haskell [18], a popular lazy functional language.

The STG language, which is the virtual machine code of the Spineless Tagless G-machine, is a very simple purely-functional language. Compared with the VMC, the Spineless Tagless G-Machine is specified at a much higher level, resembling the language to be implemented instead of a concrete machine. However, the distinguishing feature of the STG language is that it has a formal operational semantics as well as the usual denotational semantics. The operational semantics of the STG language is comparable with VMC. The machine consists five components:

- The code, similar to the code stream in the VMC.
- The argument stack, which contains a mixture of closure addresses and primitive values.
- The return stack, which contains continuations.
- The update stack, which contains update frames.
- The heap.
- The global environments, which gives the addresses of all closures defined at top level.

The mapping from the abstract machine to stock hardware, as described in [22], shows many similarities with the VMC. The three stacks are mapped into two stacks: A-stack for pointers and B-stack for non-pointers, with the name taken from the ABC machine [29]. The heap is a collection of closures of variable size. Each closure has a uniform layout: the first word of a closure is a pointer, which points to its static *info* table (not in the heap). Following the info pointer is a block of heap pointers, these are followed by a block of values other than heap pointers. This separation of heap pointers from other values is done for the same reason as in the VMC (to facilitate garbage collection). The *info* table contains a number of fields, including the address of the code to evaluate the closure (the *standard entry* code), the addresses of code to garbage collect the closure (used during garbage collection), and some other information. Different closures may share the same *info* table to save memory.

The STG machine is tagless in the sense that a closure does not contain any tags to indicate its status, such as the size of the closure, the number of pointer arguments and non-pointer arguments, whether it is in head normal form, etc. All this information is encoded either in the *info* table, or in the code segments pointed to by the *info* table. For example, the *standard entry* code for a closure that is already in head normal form is a simple return instruction.

It is also worthwhile to discuss how the STG machine updates a closure. After a closure is evaluated, it should be overwritten with the result to avoid being evaluated again, but the size of the result (also a closure) might be larger than the size of the original closure. The STG machine solves this problem by overwriting the original closure with a small *indirection* closure which contains only a *info* table pointer

and a pointer to the result. The *standard entry* code for the *indirection* closure simply loads the result closure and returns. The use of indirection closures results in extra memory allocation, but this is partially remedied since *indirection* closures are removed during garbage collection.

The major difference between the layout of closures in the STG machine and the layout of heap nodes in VMC, is that a VMC node does not have an *info* pointer. The STG machine is designed to implement *lazy* functional languages, so every closure needs a code pointer which points to the code to evaluate it. The VMC is designed to implement Charity, which we choose to implement strictly (except for coinductive data), so that code pointers are not always needed. A coinductive datum in Charity is represented as a *record* (a group of pointers, each of which points to a closure) in the VMC. The *record* serves naturally as the indirection node and so an update in the VMC does not need an indirection node.

2.4.3 The abstract machine for Standard ML of New Jersey

Standard ML of New Jersey (SML/NJ) [2] is a compiler and programming environment for the Standard ML programming language [15, 25]. The core-Charity compiler described in the following chapters has been written in this language. In his book *Compiling with Continuations* [6], A.W. Appel described an *abstract-continuation-machine*, which serves as an intermediate step in compiling SML into a particular concrete machine code.

Just like the other abstract machines, the abstract-continuation-machine has a heap for complex data. The interesting part of the machine is that it has a set of registers and does not have a stack. The CPS (continuation-passing style) trans-

formation used in the compiler ensures that a function application can never be an argument of another application, and so it is not necessary to have a stack for nested function calls. Appel argues that garbage collection can be cheaper than stack allocation [1] and so passing arguments as heap pointers can be as efficient as passing arguments on the stack.

The number of registers in the abstract machine is determined by the number of registers on target concrete machine. Most instructions in the virtual machine operate on registers only, values in registers are saved to (or loaded from) heap nodes explicitly. This makes it very easy to map the abstract machine into a concrete machine. The CPS compiler ensures that the program will not 'run out of registers' at any point by using a *register spilling* algorithm.

Another interesting aspect of the abstraction machine is how it differentiates between basic values and heap pointers: the low-order bit of each value is used as a tag to indicate whether it is a basic value or a heap pointer. This method is certainly space-efficient, but it makes arithmetic operations awkward: one cannot calculate with full 32 bit integers, and the tag bits must be stripped off operands and added to results. However, Appel estimate the cost of handling tag bits to be only 1.65 percent of total runtime in a typical SML application [3].

Overall, the *abstract-continuation-machine* is a very simple machine that can be readily mapped into a concrete machine. But the efficiency of the machine relies on the CPS translation and related optimizations. A very important property of CPS is that all arguments to functions are variables or constants, never nontrivial subexpressions. This means arguments of a function should be evaluated before being passed to the function. Thus CPS is best suited to strict functional languages.

The coinductive data in Charity is evaluated lazily which means that CPS is not directly applicable.

2.4.4 The ABC machine

The ABC machine [29, 26] is the backend for Concurrent Clean, a lazy higher-order functional language. The Concurrent Clean system is one of the fastest implementations of lazy functional languages. The name ABC comes from its three stacks:

A-stack: the stack to store pointers to heap items.

B-stack: the stack to store basic values such as integers.

C-stack: the stack to store return addresses in a subroutine call.

In addition to the three stacks, the ABC machine has a graph store (heap), a program store, a program counter, and a descriptor store. The descriptor store contains a set of descriptors, which contains information associated with symbols such as arity, instruction entry, which is called for curried applications, and the name of the symbol.

The design of the VMC borrowed the idea of having a separate pointer stack and basic stack from the ABC machine. The A-stack in the ABC machine is equivalent to the P-stack in VMC, and the B-stack in the VMC can be considered to be a combination of the B-stack and the C-stack in the ABC machine. The VMC does not have a descriptor store, because the arity and name of a symbol are handled by the compiler rather than by the virtual machine.

The major difference between the ABC and VMC machines lies in the structure of the heap nodes. As a lazy functional language, Clean was implemented as a graph reduction system. A heap node in the ABC machine has the following fields:

- A descriptor identifier which is an entry in the descriptor store. Different constructors will have different entries in the descriptor store, so this field can also be used for matching constructors.
- A code pointer which points to code that will evaluate the node to root normal form. The code pointer in a node that is already in root normal form points to a simple return instruction.
- A sequence of heap pointers that points to the arguments of the node, or, in the case that the node represents a boxed basic data, a basic value.

This node format does not allow basic values to be stored directly in a node (unless the node itself represents a boxed basic datum), because it does not provide enough information for the garbage collector to differentiate between basic values and heap pointers. This is an inefficient part of the ABC machine. For example, an integer argument of a node can not be stored directly in the node, instead, the integer has to be boxed and a pointer to the boxed integer is stored in the node. Furthermore, a boxed integer has the same format as the other heap nodes, i.e. it also contains a descriptor identifier and a code pointer, which means the cost of boxing is rather high.

The variable-sized heap node makes updating a node difficult. Unlike the STG machine, which uses an indirection node (section 2.4.2), the implementation of the ABC machine solves this problem by splitting each node into two parts: the fixed-sized part contains the descriptor id, the code pointer and a pointer to the variable-sized part. The variable-sized part contains the pointers to arguments. Now the fixed-size part of a node can always be overwritten with the fixed-size part of another

node. The downside of this solution is that it takes more space to store a node, and access to the arguments needs an extra level of indirection.

2.5 Representing Charity data in VMC

All datatypes supported by Charity can be represented efficiently in VMC. Here we will use some simple examples to show how data of different types can be created and manipulated in Charity.

2.5.1 Primitive data and tuples

Primitive datatypes, such as integers or floating point numbers, can be represented and manipulated directly on the B-stack. Most arithmetic operations, such as plus and multiply, are directly supported by VMC. For example:

```
def cube = x => x * x * x.
```

can be translated into the following VMC code:

```
    _cube:          ; x is on top of B-stack
        dupB 0      ; x
        dupB 0      ; x
        mulI        ; y=x*x
        dupB 1      ; x
        mulI        ; x*y
        retB 1 0
```

Tuples can consist a mixture of basic data and pointer data. Basic data does not need to be boxed. For example, the following code will create the tuple $(4, (5, 6))$:

```
constI 6        ; push 6 on B-stack
constI 5        ; push 5 on B-stack

newtuple 2 0    ; create tuple (5,6), pointer on P-stack

constI 4        ; push 4 on B-stack

; take 1 item from B-stack, 1 item from P-stack
; to create a tuple, which is (4,(5,6))
newtuple 1 1
```

2.5.2 Inductive Data

Each constructor of an inductive datatype will be given a constructor number, starting from 0. Inductive data will then be represented as a tuple, with the constructor number as the first field. The following definitions define the natural number datatype:

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

The constructor number of `zero` and `succ` are 0 and 1 respectively. So the data `zero` will be represented as a boxed 0, while `succ zero` will be represented as a pair: the first part of the pair is the integer 1, the second part is a pointer to a boxed 0 (a `zero`). The following VMC code will create a `succ zero`: figure 2.3 shows how it is stored in the heap.

```
constI 0       ; constructor number of zero
newtuple 1 0   ; create zero
constI 1       ; constructor number of succ
newtuple 1 1   ; create succ zero
```

Consider the `list` datatype:

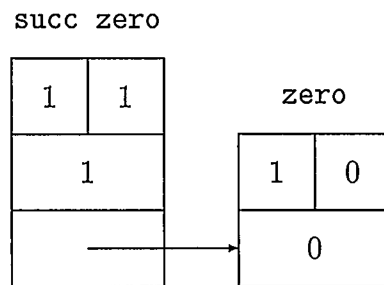


Figure 2.3: How `succ zero` is stored in the heap

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

The constructor number of `nil` and `cons` are 0 and 1 respectively. So `nil` will be represented as a boxed 0, while `cons(zero,nil)` will be represented as a triple: the first part of the triple is the integer 1, the second part is a pointer to `zero`, the third part is a pointer to `nil`. The following VMC code will create the data `cons(zero, cons(succ zero, nil))`.

```
constI 0       ; constructor number of nil
newtuple 1 0   ; create nil

constI 0       ; constructor number of zero
newtuple 1 0   ; create zero
constI 1       ; constructor number of succ
newtuple 1 1   ; create succ zero

constI 1       ; constructor number of cons
newtuple 1 2   ; create cons(succ zero, nil)

constI 0       ; constructor number of zero
newtuple 1 0   ; create zero

constI 1       ; constructor number of cons
newtuple 1 2   ; create cons(zero, cons(succ zero, nil))
```

A constructor that does not have any argument is represented by a small boxed integer, like the `zero` and `nil`. Since all the constructors such as `zero` and `nil` are represented identically and will never be updated, it is more efficient to create only one boxed 0 in the heap and use this node whenever a `zero` or `nil` is needed. The VMC provides a `newint` instruction for this optimization. `newint` *i* returns a pointer to a boxed integer *i*; the system must ensure that this boxed integer will never be updated. Currently the VMC interpreter pre-creates boxed integers 0 to 15 at the start of the heap, if the parameter of `newint` falls within this range, it will simply return the pointer to one of the pre-created nodes, otherwise, a new boxed integer will be created. In some cases, this seemingly small optimization can actually reduce heap usage by over 25 percent. Using the `newint` instruction, the above code sequence becomes:

```
newint 0       ; create nil

newint 0       ; create zero
constI 1       ; constructor number of succ
newtuple 1 1   ; create succ zero

constI 1       ; constructor number of cons
newtuple 1 2   ; create cons(succ zero, nil)

newint 0       ; create zero

constI 1       ; constructor number of cons
newtuple 1 2   ; create cons(zero, cons(succ zero, nil))
```

2.5.3 Coinductive data representation

Coinductive data with n destructors can be represented in VMC as a n-tuple of *closure* pointers, with each *closure* corresponding to a destructor. A *closure* is a

tuple with its first field being a code label (address), and the other fields being the arguments of the code.

When a destructor is applied to a coinductive data, the destructor fetches the corresponding closure from the coinductive data then evaluates that closure. To evaluate a closure, a simple `detuple` instruction will push everything in the closure onto the stacks, with the code label on top of the B-stack. Thereupon, a simple `xcall` instruction will run the code at that label and remove it from the stack.

Here we use a simple coinductive datatype `Lprod` as an example.

```
data C -> Lprod = fst: C -> int
                | snd: C -> int.
```

The code generated for destructors is very short, so it is generated as inline code instead of as a function call. The code generated for destructor `fst` and `snd` will looks like this:

```
;code generated for destructor fst
dupP 0      ; to avoid the record pointer being popped off by getfield
getfieldP 0 ; get the first closure in the record
detuple     ; put everything in the closure in stack
xcall       ; call the associated code

;code generated for destructor snd
dupP 0      ; to avoid the record pointer being popped off by getfield
getfieldP 1 ; get the second pointer in the record
detuple     ; put everything in the closure in stack
xcall       ; call the associated code
```

When evaluating a closure, the record that contains the pointer to the closure will be passed to the code associated with the closure, so that the record can be updated. Upon return from the code, the result of the computation should be on top of the B-stack or P-stack.

The code to create a record is divided into two parts: the first part is the code associated with each closure, the second part is the code to create closures and tuples. For example, to create a record `(fst:3, snd:5)`, the code associated with each closure must be produced first. In this case, a simple return instruction is sufficient because neither 3 nor 5 needs further evaluation. The result is stored directly in the closure, so that at entry of the following closure code, the result (3 or 5) is already on top of the B-stack.

```
_rec35fst:      ; code entry for the first closure
    retB 0 1    ; pop off the record pointer and return
_rec35snd:      ; code entry for the second closure
    retB 0 1    ; pop off the record pointer and return
```

Now we show the code to create the closures and the record.

```
constI 5           ;
constA _rec35snd   ; code entry
newtuple 2 0       ; create the closure for snd

constI 3           ;
constA _rec35fst   ; code entry
newtuple 2 0       ; create the closure for fst

newtuple 0 2       ; create the record
```

Figure 2.4 shows the how the record (fst:3, snd:5) is stored in the heap.

A slightly more complicated example illustrates how non-trivial closure are created:

```
def foo = x => (fst:x+x, snd:x*x).
```

will be translated into:

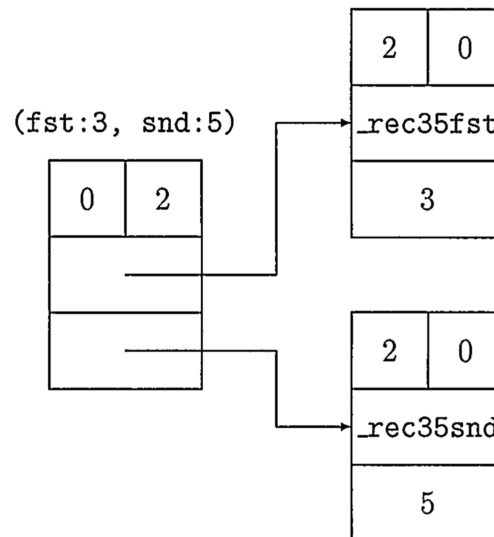


Figure 2.4: How (fst:3, snd:5) is stored in the heap

```
_foofst:              ; code entry for the first closure

    dupB 0            ; x is on top of B-stack
    addI              ; x+x

    ;create a new closure to store the result
    dupB 0            ; result
    constA _fooret    ; code entry for the new closure
    newtuple 2 0      ; create the closure

    setfieldP 0       ;update the record with the new closure
    ret

_foosnd:              ; code entry for the second closure

    dupB 0            ; x is on top of B-stack
    mulI              ; x*x

    ;create a new closure to store the result
    dupB 0            ; result
    constA _fooret    ; code entry for the new closure
```

```
    newtuple 2 0    ; create the closure

    setfieldP 1     ;update the record with the new closure
    ret

_fooret:
    retB 0 1

_foo:               ; code entry for function foo

    dupB 0          ; x, argument for the second closure
    constA _foosnd  ; code entry for the second closure
    newtuple 2 0    ; create the second closure

    dupB 0          ; x, argument for the first closure
    constA _foofst  ; code entry for the first closure
    newtuple 2 0    ; create the first closure

    newtuple 0 2    ; create the record

    retP 1 0
```

In the above code, x is stored in the closures as a parameter, the code associated with the closures will compute x+x and x*x respectively and update the record.

More complicated coinductive data, such as those created by an *unfold*, can always be translated into a *record* (section 3.2), then be translated into VMC code in a manner similar to the above. For higher-order destructors, things are even simpler: the result of the computation does not need to be saved, as different parameters to the destructor will give different results. Hence, it is not necessary to build a new closure and update the record (section 4.2).

Chapter 3

Intermediate representation for Charity code

To translate Charity code into VMC code, we go through the following steps:

1. Lexical and syntactic analysis: this is achieved with the help of the lex and yacc tools of SML. The result of this step is a syntax tree.

2. Semantic checks: this step checks the syntax tree to find any semantic errors. The most important component of semantic checking is type-checking. This follows the method described in [33]. The syntax tree is passed to the next step if no error is found, otherwise an error message is given and the translation stops.

3. Translation into an intermediate representation (IR): this step translates the syntax tree into the internal representation of Charity program (IR). This extra level of translation makes the compiler better structured, and, furthermore, this internal representation facilitates manipulations.

4. Transformations of the IR: Combinator invocations are transformed into normal function calls, and different versions of polymorphic functions are generated to handle unboxed data efficiently.

5. Translation of the IR into VMC code: the simplified IR is translated into VMC code by several translation rules.

This chapter describes the IR and the transformations over it. The next chapter will describe the translation from IR to VMC code.

3.1 Structure of Intermediate Representation

3.1.1 Representation of type expressions and multi-functions

We shall explain the structure of the IR using SML datatypes. The description is thus very close to the actual SML code. A Charity type expression can be:

- A type variable, which can match any type, e.g. A
- A type's name, with or without arguments, e.g. $list(A)$
- Products of several types, e.g. $list(A) * A * int$

The following ML datatype is a representation of a Charity type expression:

```
datatype TypeExp
       = tVAR of int                       (* Type Variable *)
       | tNAME of string * TypeExp list    (* Type name *)
       | tPRODS of TypeExp list            (* Products*)
```

Here each type variable is distinguished by an integer. If there are n type variables in a type expression, we indicate each with an integer from 1 to n. For example, $list(A) * B$ with be represented as

```
tPRODS [tNAME("list",[tVAR 1]), tVAR 2]
```

The type of a Charity function is an arrow from a list of type expressions to another type expression, e.g. $A, list(A) \to B$, which can be represented by the following ML type:

```
type FunType = TypeExp list * TypeExp
```

The new implementation extends the syntax of the arrow type to allow multiple inputs (a "multi-function"):

$$S_1, \ldots, S_n \rightarrow T$$

where S_i and T are any type expressions except arrow types. The purpose of this new syntax is to specify precisely how the arguments are passed to the function. For example, the following two functions f and g have distinct types:

```
def f: int , list(int) -> int = ...
def g: int * list(int) -> int = ...
```

f has two arguments, while g has only one argument which is a pair. The following figure shows the different stack status at the entry of f and g while calling $f(2, [4, 5])$ and $g(2, [4, 5])$.

Code	B-stack	P-stack	Heap
$f(2, [4, 5])$	$2.bs$	$p.ps$	$p \rightarrow [4\ 5]$
$g(2, [4, 5])$	bs	$p'.ps$	$p' \rightarrow (2, [4\ 5])$

The type of f is expressed in ML as

```
([tNAME("int",[]),
  tNAME("list",[tNAME("int",[])])
 ],
 tNAME("int",[]))
```

while type of g will be expressed as

```
([tPRODS [tNAME("int",[]),
                tNAME("list",[tNAME("int",[])])
              ]
 ],
 tNAME("int",[]))
```

The new syntax also controls how inductive data is stored in the heap. As an example, consider the following two definitions of the list datatype:

```
data list1 -> C =  nil1: 1 -> C
                | cons1: int,C -> C.

data list2 -> C =  nil2: 1 -> C
                | cons2: int*C -> C.
```

Here `cons1` takes two arguments: an integer and a list. `cons2` takes only one argument: a pair that consists of an integer and a list. Figure 3.1 shows how `cons1(5,nil1)` and `cons2(5,nil2)` are stored in the heap. The definition of `list1` is obviously more space-efficient, but there are cases where `list2` is more desirable.

The definition of TypeExp makes it easy to analyze and transform type expressions in a program, but more difficult to read. So in the following sections, type expressions are represented in a way very similar to that in Charity, except that Tn is used to represent type variable n. For example, $list(A), B \to A$ will be expressed as $list(T1), T2 \to T1$.

3.1.2 Symbol Table

The symbol table is a list of pairs, the first element of the pair is a symbol, and the second element of the pair is the property of the symbol, which indicate the type

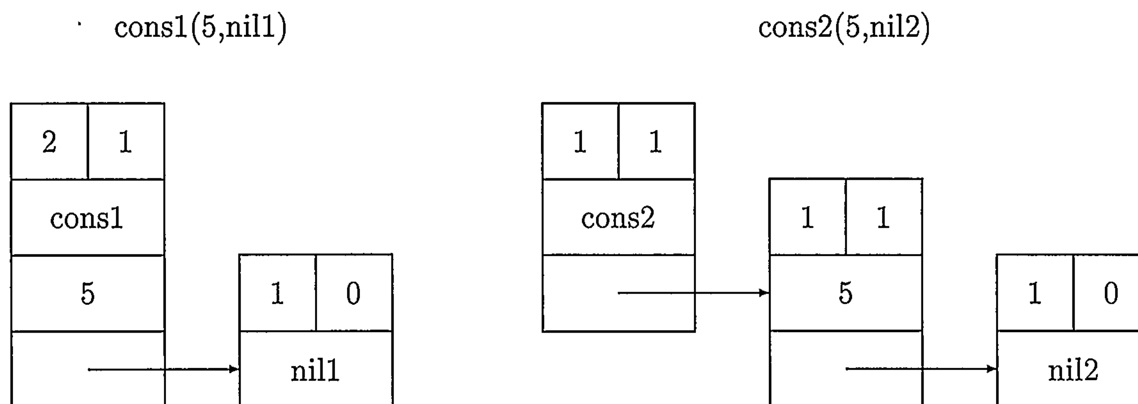


Figure 3.1: cons1 vs. cons2

of the symbol and its corresponding values. We use the following ML datatype to represent properties of symbols:

```
datatype SymProperty
   = ALIAS  of int * TypeExp                      (* Alias       *)
   | DATA   of int * ((string*TypeExp) list)      (* Inductive   *)
   | CODATA of int * ((string*TypeExp) list)      (* Coinductive *)
   | CTR    of string * int * TypeExp             (* Constructor *)
   | DTR    of string * int * TypeExp             (* Destructor  *)
   | FUN    of TypeExp * TypeExp list * Term      (* Function    *)
```

The properties of an alias are the number of type variables that it has and the type expression it represents. For example:

```
data listlist(A) = list(list(A))
```

will be translated into the symbol table entry:

```
("listlist",  ALIAS(1, list(list(T1))) )
```

Here we indicate that we defined a type alias named "listlist", it has one type variable T_1, and is equivalent to the type $list(list(T_1))$.

The properties of an inductive or coinductive datatype definition are the number of type variables in the datatype definition and a list of constructor/destructor names and constructor/destructor types. When translating the structor types, the state variable C is translated into a special type variable, T0. For example:

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

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

Will be translated into

```
("list",    DATA(1, [ ( "nil", ( -> T0) ),
                      ("cons", (T1,T0 -> T0))]))

("inflist", CODATA(1, [ ("head", (T0 -> T1)),
                        ("tail", (T0 -> T0))]))
```

The properties of a constructor/destructor are: the name of the datatype, the number of the structor and the type of the structor. A structor is treated as a function in type checking, and so T0 in the type of the structor is replaced with the data name. In the last example, the following entries will be added to the symbol table:

```
( "nil",  CTR("list", 0, ( -> list(T1))))

("cons",  CTR("list", 1, ( T1,list(T1)  -> list(T1) )))

("head",  DTR("list", 0, ( inflist(T1) -> T1)))

("tail",  DTR("tail", 1, ( inflist(T1) -> inflist(T1))))
```

The properties of a function definition are the type of the function, the types of the macros, and the function's body. We use the following ML datatype to represent the body of a function:

```
datatype Term
   = INT of integer                (* integer constant *)
   | STR of string                 (* string constant *)
   | VAR of string                 (* variable *)
   | TUPLE of Term list            (* n-tuple, including 0-tuple*)
   | APP of Term * Term            (* application *)
   | RECORD of string * Term list  (* record *)
   | CASE of string * Term list    (* case *)
   | MACRO of int                  (* macro *)
   | CTR of string * int           (* constructor *)
   | DTR of string * int           (* destructor  *)
   | CALL of string * Term list    (* call a function *)
   | ABS of string list * Term     (* abstraction *)
```

A *record* in Charity will be represented by a RECORD, which consists of the data type's name and a list of record phrases. The list of record phrases are sorted to match the order of destructors declared in the data definition. A *case* in Charity is similarly represented by a CASE. Each *macro* is given a number starting from 1, in their declaration order. An abstraction is composed of a list of variable names and a term. Here is an example:

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

By type inference, the types of f and g are determined to be $T1 \to T1 * T2$ and $T1 \to T2$, respectively. The body of f is translated into:

```
ABS(["x"], TUPLE [VAR "x", APP(MACRO 1, VAR "x")])
```

Here macro g is translated into MACRO 1. The resulting entry in the symbol table is

```
("f", FUN( T1 -> T1*T2, [ T1 -> T2 ],
          ABS(["x"], TUPLE [ VAR "x", APP(MACRO 1, VAR "x")]))
)
```

For clarity, in this thesis, a term is represented in a form similar to that of Charity code, as in Table 3.1. For example, the body of f in the above example will be represented as

$$x \Rightarrow (x, \mathrm{m}_1\ x)$$

3.2 Fold, Unfold and Map

There are no structures which correspond directly to *fold*, *unfold* and *map*. Instead, *fold*, *unfold* and *map* can be expressed with (possibly recursive) *case/record*. For an inductive datatype D, the system automatically generates two combinators *fold_D* and *map_D*. For a coinductive datatype D, *unfold_D* and *map_D* are generated. Then a *fold/unfold/map* will simply be translated into a *combinator invocation*. For example, if we define list as

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

The compiler will generate two combinators named *fold_list* and *map_list*. Where *fold_list* has two macros which correspond to the two fold phrases, *map_list* has one macro corresponding to one map phrase. So the body of the function

```
def app = L1,L2 => {|  nil: => L2
                     | cons: a,l => cons(a,l)
                     |} L1.
```

will be translated into

$$L1, L2 \Rightarrow \mathit{fold_list} \left\{ \begin{array}{c} \Rightarrow L2 \\ a, l \Rightarrow c_1^{list}\ (a, l) \end{array} \right\} L1$$

Formally, an inductive datatype D is defined as

$$\text{data } D(A_1, \ldots, A_m) \rightarrow C = \left| \begin{array}{lll} c_1 & : & E_1^1, E_1^2, \ldots, E_1^{k_1} \rightarrow C \\ c_2 & : & E_2^1, E_2^2, \ldots, E_2^{k_2} \rightarrow C \\ \vdots & & \\ c_n & : & E_n^1, E_n^2, \ldots, E_n^{k_n} \rightarrow C \end{array} \right.$$

Where A_i is a type variable, C is the state variable, k_i is the number of arguments of constructor i. E_i^j is a type expression to specify the type of constructor i's jth argument. E_i^j is defined as:

$$\begin{array}{rcll} E & ::= & A_i & \text{type variable} \\ & | & C & \text{state variable} \\ & | & E_1 * E_2 \cdots * E_k & \text{product} \\ & | & T(E_1, \ldots, E_k) & \text{datatype} \end{array}$$

Given a type expression E as defined above, we generate a combinator $\theta^E\{f, g_1, \ldots, g_m\}$, where θ^E takes an input x of type E, maps f over the state variable part (the recursive part) of x, and maps g_i over the type variable A_i part of x. We use the following

rules to recursively generate the body of θ^E:

$$\begin{aligned}
\theta^{A_i} &= g_i \\
\theta^{C} &= f \\
\theta^{E_1 * \cdots * E_k} &= \{v_1, \ldots, v_k \Rightarrow (\theta^{E_1} v_1, \ldots, \theta^{E_k} v_k)\} \\
\theta^{T(E_1, \ldots, E_k)} &= map_T\{\theta^{E_1}, \ldots, \theta^{E_k}\}
\end{aligned}$$

Then *fold_D* and *map_D* can be defined as:

$$fold_D\{g_1, \ldots, g_n\} = \text{case}^D \left\{ \begin{array}{lcl} v_1, v_2, \ldots, v_{k_1} & \Rightarrow & g_1\ (F_1^1\ v_1, \ldots, F_1^{k_1}\ v_{k_1}) \\ v_1, v_2, \ldots, v_{k_2} & \Rightarrow & g_2\ (F_2^1\ v_1, \ldots, F_2^{k_2}\ v_{k_2}) \\ \vdots & & \\ v_1, v_2, \ldots, v_{k_n} & \Rightarrow & g_n\ (F_n^1\ v_1, \ldots, F_n^{k_n}\ v_{k_n}) \end{array} \right\}$$

where $F_i^j = \theta^{E_i^j}\{fold_D, I_1, \ldots, I_m\}$, I_i is the identity abstraction which can be defined as $\{x \Rightarrow x\}$.

$$map_D\{g_1, \ldots, g_m\} = \text{case}^D \left\{ \begin{array}{lcl} v_1, v_2, \ldots, v_{k_1} & \Rightarrow & \text{c}_1^D\ (G_1^1\ v_1, \ldots, G_1^{k_1}\ v_{k_1}) \\ v_1, v_2, \ldots, v_{k_2} & \Rightarrow & \text{c}_2^D\ (G_2^1\ v_1, \ldots, G_2^{k_2}\ v_{k_2}) \\ \vdots & & \\ v_1, v_2, \ldots, v_{k_n} & \Rightarrow & \text{c}_n^D\ (G_n^1\ v_1, \ldots, G_n^{k_n}\ v_{k_n}) \end{array} \right\}$$

where $G_i^j = \theta^{E_i^j}\{map_D, g_1, \ldots, g_m\}$.

For example, the *list* datatype is defined as

$$\text{data } list(A) \to C = \left| \begin{array}{lcl} nil & : & \to C \\ cons & : & A, C \to C \end{array} \right.$$

Here $n = 2$, $m = 1$, $k_1 = 0$ and $k_2 = 2$, so we get

$$fold_list\{g_1, g_2\} = \text{case}^{list}\left\{ \begin{array}{c} \Rightarrow g_1() \\ v_1, v_2 \Rightarrow g_2(F_2^1\ v_1, F_2^2\ v_2) \end{array} \right\}$$

$$map_list\{g\} = \text{case}^{list}\left\{ \begin{array}{c} \Rightarrow \mathrm{c}_1^{list}() \\ v_1, v_2 \Rightarrow \mathrm{c}_2^{list}(G_2^1\ v_1, G_2^2\ v_2) \end{array} \right\}$$

where

$$\begin{array}{lclclcl} F_2^1 & = & \theta^{E_2^1}\{fold_list, I\} & = & \theta^{A}\{fold_list, I\} & = & I \\ F_2^2 & = & \theta^{E_2^2}\{fold_list, I\} & = & \theta^{C}\{fold_list, I\} & = & fold_list \\ G_2^1 & = & \theta^{E_2^1}\{map_list, g\} & = & \theta^{A}\{map_list, g\} & = & g \\ G_2^2 & = & \theta^{E_2^2}\{map_list, g\} & = & \theta^{C}\{map_list, g\} & = & map_list \end{array}$$

Substituting the above equations into the body of *fold_list* and *map_list*, we get

$$fold_list\{g_1, g_2\} = \text{case}^{list}\left\{ \begin{array}{c} \Rightarrow g_1() \\ v_1, v_2 \Rightarrow g_2(I\ v_1, fold_list\ v_2) \end{array} \right\}$$

$$map_list\{g\} = \text{case}^{list}\left\{ \begin{array}{c} \Rightarrow \mathrm{c}_1^{list}() \\ v_1, v_2 \Rightarrow \mathrm{c}_2^{list}(g\ v_1, map_list\ v_2) \end{array} \right\}$$

Note that although both *fold_list* and *map_list* are recursive, we write *fold_list* instead of *fold_list*$\{g_1, g_2\}$ for the recursive call, since the macros are always the same, and because it simplifies the work below (section 3.3), since the macros in the recursive calls need not be substituted out.

Coinductive datatypes are defined as follows:

$$\text{data } C \to D(A_1, \ldots, A_m) = \left| \begin{array}{lll} d_1 & : & H_1^1, H_1^2, \ldots, H_1^{k_1}, C \to E_1 \\ d_2 & : & H_2^1, H_2^2, \ldots, H_2^{k_2}, C \to E_2 \\ \vdots & & \\ d_n & : & H_n^1, H_n^2, \ldots, H_n^{k_n}, C \to E_n \end{array} \right.$$

Here n is the number of destructors, m is the number of type variables, and k_i is the number of higher-order arguments of destructor d_i (d_i is not higher-order if $k_i = 0$). H_i^j is the type of d_i's jth higher order argument, E_i is d_i's result type. H_i^j and E_i are defined in the same manner as the E_i^j in the inductive datatype. Note that a higher-order destructor declared in the form

$$d_j : C \to H_j^1, \ldots, H_j^{k_j} \Rightarrow E_j$$

is just syntactic sugar: the system actually reads it as

$$d_j : H_j^1, \ldots, H_j^{k_j}, C \to E_j$$

unfold_D and *map_D* are generated as:

$$
\begin{aligned}
& \mathit{unfold_D}\{g_1, \ldots, g_n\} \\
= \quad & x \Rightarrow \mathrm{rec}^D \left\{ \begin{array}{lcl}
v_1, \ldots, v_{k_1} & \Rightarrow & \theta^{E_1}\{\mathit{unfold_D}, I_1, \ldots, I_m\}\ g_1(v_1, \ldots, v_{k_1}, x) \\
v_1, \ldots, v_{k_2} & \Rightarrow & \theta^{E_2}\{\mathit{unfold_D}, I_1, \ldots, I_m\}\ g_2(v_1, \ldots, v_{k_2}, x) \\
\vdots & & \\
v_1, \ldots, v_{k_n} & \Rightarrow & \theta^{E_n}\{\mathit{unfold_D}, I_1, \ldots, I_m\}\ g_n(v_1, \ldots, v_{k_n}, x)
\end{array} \right\}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{map_D}\{g_1, \ldots, g_m\} \\
= \quad & x \Rightarrow \mathrm{rec}^D \left\{ \begin{array}{lcl}
v_1, \ldots, v_{k_1} & \Rightarrow & \theta^{E_1}\{\mathit{map_D}, g_1, \ldots, g_m\}\ \mathrm{d}_1^D(v_1, \ldots, v_{k_1}, x) \\
v_1, \ldots, v_{k_2} & \Rightarrow & \theta^{E_2}\{\mathit{map_D}, g_1, \ldots, g_m\}\ \mathrm{d}_2^D(v_1, \ldots, v_{k_2}, x) \\
\vdots & & \\
v_1, \ldots, v_{k_n} & \Rightarrow & \theta^{E_n}\{\mathit{map_D}, g_1, \ldots, g_m\}\ \mathrm{d}_n^D(v_n, \ldots, v_{k_n}, x)
\end{array} \right\}
\end{aligned}
$$

Note that $unfold$ in the form of

$$
\left(\left| x \Rightarrow \begin{array}{lcl}
\vdots & & \\
d_i & : & \tau_i \\
\vdots & & \\
d_j & : & v_j \Rightarrow \tau_j \\
\vdots & &
\end{array} \right|\right)
$$

will be translated into

$$
\mathit{unfold_D}\{\ldots, \{x \Rightarrow \tau_i\}, \ldots, \{v_j, x \Rightarrow \tau_j\}, \ldots\}
$$

Here are two examples:

1. We define the datatype for infinite list as

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

Here $n = 2$, $m = 1$, $k_1 = k_2 = 0$, Then the $unfold$ and map will be generated as

$$unfold_IL\{g_1, g_2\} = x \Rightarrow \text{rec}^{IL}\left\{\begin{array}{l} \Rightarrow\ \theta^{E_1}\{unfold_IL, I\}\ g_1\ x \\ \Rightarrow\ \theta^{E_2}\{unfold_IL, I\}\ g_2\ x \end{array}\right\}$$

$$map_IL\{g\} = x \Rightarrow \text{rec}^{IL}\left\{\begin{array}{l} \Rightarrow\ \theta^{E_1}\{map_IL, g\}\ \text{d}_1^{IL}\ x \\ \Rightarrow\ \theta^{E_2}\{map_IL, g\}\ \text{d}_2^{IL}\ x \end{array}\right\}$$

Since $\theta^{E_1}\{f, g\} = \theta^{A}\{f, g\} = g$ and $\theta^{E_2}\{f, g\} = \theta^{C}\{f, g\} = f$, we get

$$unfold_IL\{g_1, g_2\} = x \Rightarrow \text{rec}^{IL}\left\{\begin{array}{l} \Rightarrow\ I\ g_1\ x \\ \Rightarrow\ unfold_IL\ g_2\ x \end{array}\right\}$$

$$map_IL\{g\} = x \Rightarrow \text{rec}^{IL}\left\{\begin{array}{l} \Rightarrow\ g\ \text{d}_1^{IL}\ x \\ \Rightarrow\ map_IL\ \text{d}_2^{IL}\ x \end{array}\right\}$$

2. The exponential which allows the use of higher-order functions is defined as

$$\text{data } C \to exp(A, B) = fn : C \to A \Rightarrow B$$

For clarity, we rewrite it as

$$\text{data } C \to exp(A_1, A_2) = fn : A_1, C \to A_2$$

So $n = 1$, $m = 2$, $k_1 = 1$.

$$unfold_exp\{g\} = x \Rightarrow \mathrm{rec}^{exp}\Big\{v \Rightarrow \theta^{E_1}\{unfold_exp, I, I\}\ g(v, x)\Big\}$$

$$map_exp\{g_1, g_2\} = x \Rightarrow \mathrm{rec}^{exp}\Big\{v \Rightarrow \theta^{E_1}\{map_exp, g_1, g_2\}\ \mathrm{d}_1^{exp}(v, x)\Big\}$$

Here $\theta^{E_1}\{f, g_1, g_2\} = \theta^{A_2}\{f, g_1, g_2\} = g_2$, so

$$unfold_exp\{g\} = x \Rightarrow \mathrm{rec}^{exp}\{v \Rightarrow I\ g(v, x)\}$$

$$map_exp\{g_1, g_2\} = x \Rightarrow \mathrm{rec}^{exp}\{v \Rightarrow g_2\ \mathrm{d}_1^{exp}(v, x)\}$$

3.3 Transforming combinator invocations

3.3.1 Introduction

Charity does not support higher-order functions explicitly (see example 2 above). However, Charity gives explicit support for the use of combinators. Combinators can be used to obtain some of the functionality of higher-order functions. There are at least two possible ways to implement combinators:

1. Compile macros in combinators as higher-order functions. That is, each macro is an actual argument of the combinator and its value is passed on the P-stack in the form of a closure, which is composed of a code pointer and its environment. This method usually yields smaller code size than method 2, however, the cost of building/decomposing closures usually will mean slower execution speed and higher heap usage.

2. Compile macros in combinators as *macros* in the original meaning of the word. That is, multiple new functions are generated, each corresponding to a combinator invocation with different macro arguments, with a combinator invocation transformed into a call to the corresponding new function. A detailed explanation follows. Obviously, code size will usually be larger–possibly exponentially larger (see discussion in section 3.3.4)– than in method 1, but since no closures need to be created and passed as arguments, the execution speed will generally be faster and heap usage will be lower.

Method 2 is attractive as the benefit of faster code and lower heap usage is usually higher than that gained from smaller code size. Furthermore, we can always define the *exponential* datatype in Charity:

$$\text{data } C \to exp(A, B) = fn : C \to A \Rightarrow B.$$

Charity then effectively accepts higher-order functions that can be passed as parameters in the form of records of type *exp*. So if the user prefers smaller code size, they can always use the *exp* datatype instead of macros. If execution speed is more important, then the user can choose macros. Thus method 2 promises to give the user more control. However, there is a problem: this sometimes leads to a code explosion. So we actually implement a hybrid method as explained in section 3.3.4.

Here is a simple example to show the basic idea of method 2.

$$\begin{aligned} f\{g\} &= x \Rightarrow (x, g\,x) \\ h &= x \Rightarrow f\{y \Rightarrow (y, y)\}x \end{aligned}$$

Here f is a combinator, h is not a combinator, but it contains a combinator invocation $f\{y \Rightarrow (y,y)\}$. To translate combinator invocation in h, we generate a new function f' by substituting g with $y \Rightarrow (y,y)$ in the body of f, then substitute $f\{y \Rightarrow (y,y)\}$ in h with f', and we get

$$\begin{aligned} f' &= x \Rightarrow (x, \{y \Rightarrow (y,y)\}\ x) \\ h &= x \Rightarrow f'\ x \end{aligned}$$

However, if we change the above example to

$$\begin{aligned} f\{g\} &= x \Rightarrow (x, g\ x) \\ h &= x, y \Rightarrow f\{z \Rightarrow (y,z)\}x \end{aligned}$$

After similar substitutions, we get

$$\begin{aligned} f' &= x \Rightarrow (x, \{z \Rightarrow (y,z)\}\ x) \\ h &= x, y \Rightarrow f'\ x \end{aligned}$$

Now a problem arises: we have an undefined variable y in f'. To solve this, we add a parameter y to f', and change the call to f' correspondingly so that f' takes two arguments. This technique is called λ-lifting [20].

$$\begin{aligned} f' &= x, y \Rightarrow (x, \{z \Rightarrow (y,z)\}\ x) \\ h &= x, y \Rightarrow f'\ (x,y) \end{aligned}$$

The above example shows the basic idea of transforming combinators invocations into normal function calls. We describe the general steps in the following sections

3.3.2 Environment variables

First, we define the notion of an environment variable. We say a variable v is an environment variable of a term τ if v occurs in τ but is not defined in τ. The set of all environment variables of τ is called the environment of τ, denote as $\text{env}\langle\tau\rangle$. To find the environment variables of τ, we apply the following rules recursively:

- Variable:

$$\text{env}\langle x\rangle = \{x\}$$

- Tuple:

$$\text{env}\langle(\tau_1, \ldots, \tau_n)\rangle = \text{env}\langle\tau_1\rangle \cup \cdots \cup \text{env}\langle\tau_n\rangle$$

- Abstraction:

$$\text{env}\langle v_1, \ldots, v_n \Rightarrow \tau\rangle = \text{env}\langle\tau\rangle - \{v_1, \ldots, v_n\}$$

- Function call:

$$\text{env}\langle f\{\tau_1, \ldots, \tau_n\}\rangle = \text{env}\langle\tau_1\rangle \cup \cdots \cup \text{env}\langle\tau_n\rangle$$

- Case:

$$\text{env}\langle\text{case}^D\{\tau_1, \ldots, \tau_n\}\rangle = \text{env}\langle\tau_1\rangle \cup \cdots \cup \text{env}\langle\tau_n\rangle$$

- Record:

$$\text{env}\langle\text{rec}^D\{\tau_1, \ldots, \tau_n\}\rangle = \text{env}\langle\tau_1\rangle \cup \cdots \cup \text{env}\langle\tau_n\rangle$$

- Application:

$$\text{env}\langle \tau_1\ \tau_2 \rangle = \text{env}\langle \tau_1 \rangle \cup \text{env}\langle \tau_2 \rangle$$

- Other:

$$\text{env}\langle \tau \rangle = \emptyset$$

Given an abstraction with environment variables, we can transform it into an abstraction without environment variables by adding those environment variables as new parameters of the abstraction. For example, $x \Rightarrow x + y$ has an environment variable y, by adding y as a new parameter of the abstraction, we get $x, y \Rightarrow x + y$ which has no environment variables.

The following rules will transform a term with environment variables into a term without environment variables.

- Abstractions:

$$\text{lift}\langle v_1, \ldots, v_n \Rightarrow \tau \rangle = v_1, \ldots, v_n, e_1, \ldots, e_k \Rightarrow \tau$$

  where $e_1, \ldots, e_k = \text{env}\langle v_1, \ldots, v_n \Rightarrow \tau \rangle$.

- All other terms:

$$\text{lift}\langle \tau \rangle = e_1, \ldots, e_k \Rightarrow \tau$$

  where $e_1, \ldots, e_k = \text{env}\langle \tau \rangle$.

3.3.3 Generating new functions

To translate combinators into normal functions, the basic idea is to generate a new function for each combinator invocation, then a combinator invocation can be substituted by a call to the new function.

First, we discuss how to generate a new function f' which is equivalent to $f\{\tau_1, \ldots, \tau_n\}$, where f is defined as $v_1, \ldots, v_m \Rightarrow \tau$. The naive approach is to simply substitute the macros $\mathrm{m}_1, \ldots, \mathrm{m}_n$ in τ with actual macro arguments $\tau_1, \ldots, \tau_n$, that is, $f' = v_1, \ldots, v_m \Rightarrow \tau'$, where $\tau' = \tau[\tau_i/\mathrm{m}_i]_{i=1}^{n}$. However, we have to consider the following problems:

1. If $\tau_1, \ldots, \tau_n$ contains unbound variables, the term $v_1, \ldots, v_m \Rightarrow \tau'$ will contain unbound variables as well. The solution is to use λ-lifting as given in the previous section:

$$f' = \mathrm{lift}\langle v_1, \ldots, v_m \Rightarrow \tau'\rangle = v_1, \ldots, v_m, e_1, \ldots, e_k \Rightarrow \tau'.$$

   where $e_1, \ldots, e_k = \mathrm{env}\langle f\{\tau_1, \ldots, \tau_n\}\rangle$.

2. If f is a recursive combinator (as happens, for example, with the *fold_D* generated by the system), then an invocation of f within τ should be replaced with a call to f'. However, the lifted function f' may have more arguments than f. This means we cannot simply substitute f with f' due to the type mismatch. The solution is to substitute f with the following abstraction:

$$v_1, \ldots, v_m \Rightarrow f'(v_1, \ldots, v_m, e_1, \ldots, e_k)$$

The above abstraction takes the same number of arguments as f, and calls f' with the correct number of arguments as well. Obviously its type matches that of f.

3. $\tau_1, \ldots, \tau_n$ may contain an unbound variable which has the same name as a variable defined in f, in which case, the unbound variable will be mistakenly captured as a local variable defined in f. The solution is to rename all the unbound variables $e_1, \ldots, e_k$ to new variables $e'_1, \ldots, e'_k$ before the substitution, where $e'_1, \ldots, e'_k$ are system generated variable names that are all guaranteed to be unique.

After f' is generated, we can generate the term to substitute $f\{\tau_1, \ldots, \tau_n\}$. Similar to the problem discussed above, we cannot simply use f' as the substitute. Instead, we use the following abstraction to substitute $f\{\tau_1, \ldots, \tau_n\}$:

$$w_1, \ldots, w_m, e_1, \ldots, e_k \Rightarrow f'\ (w_1, \ldots, w_m, e_1, \ldots, e_k)$$

where $w_1, \ldots, w_m$ are new variables distinct from $e_1, \ldots, e_k$.

This gives the following translation rule, $G[\![-]\!]$, for translating a combinator invocation into a call to a new function: Let $f\{\mathrm{m}_1, \ldots, \mathrm{m}_n\} = v_1, \ldots, v_m \Rightarrow \tau$, then

$$G[\![f\{\tau_1, \ldots, \tau_n\}]\!] = w_1, \ldots, w_m, e_1, \ldots, e_k \Rightarrow f'\{(w_1, \ldots, w_m, e_1, \ldots, e_k)\}$$

where

$$\{e_1, \ldots, e_k\} = \mathrm{env}\langle\tau_1\rangle \cup \cdots \cup \mathrm{env}\langle\tau_n\rangle$$

and $w_1, \ldots, w_m$ are new variables distinct from $e_1, \ldots, e_k$. f' is the name of a new function defined as:

$$v_1, \ldots, v_m, e'_1, \ldots, e'_k \Rightarrow \tau[\eta/f][\tau'_i/\mathrm{m}_i]_{i=1}^{n}$$

where $e'_1, \ldots, e'_k$ are new variables corresponding to $e_1, \ldots, e_k$ and

$$\begin{aligned} \tau'_i &= \tau_i[e'_j/e_j]_{j=1}^{k} \\ \eta &= \{v_1, \ldots, v_m \Rightarrow f'\{(v_1, \ldots, v_m, e'_1, \ldots, e'_k)\} \end{aligned}$$

The following rules, $M[\![-]\!]$, will transform a term with combinator invocations into a term without combinator invocations:

combinator invocation	$M[\![f\{\tau_1, \ldots, \tau_n\}]\!] = G[\![f\{M[\![\tau_1]\!], \ldots, M[\![\tau_n]\!]\}]\!]$
abstraction	$M[\![v_1, \ldots, v_n \Rightarrow \tau]\!] = v_1, \ldots, v_n \Rightarrow M[\![\tau]\!]$
tuple	$M[\![(\tau_1, \ldots, \tau_n)]\!] = (M[\![\tau_1]\!], \ldots, M[\![\tau_n]\!])$
case	$M[\![\mathrm{case}^D\{\tau_1, \ldots, \tau_n\}]\!] = \mathrm{case}^D\{M[\![\tau_1]\!], \ldots, M[\![\tau_n]\!]\}$
record	$M[\![\mathrm{rec}^D\{\tau_1, \ldots, \tau_n\}]\!] = \mathrm{rec}^D\{M[\![\tau_1]\!], \ldots, M[\![\tau_n]\!]\}$
application	$M[\![\tau_1\ \tau_2]\!] = M[\![\tau_1]\!]\ M[\![\tau_2]\!]$
other	$M[\![\tau]\!] = \tau$

The new functions generated by the G rules may also contain function calls with macros, which should also be transformed by applying the M rules.

3.3.4 Avoiding code explosion

Note the M rules for a function call $f\{\tau_1, \ldots, \tau_n\}$: it applies the M rules on τ_i before applying the G rules. So the inner-most combinator invocations will be transformed first. This transformation order helps to reduce the number of newly generated functions. Consider the following example:

$$\begin{aligned} f\{\mathrm{m}\} &= x \Rightarrow x + (\mathrm{m}\ x) \\ g\{\mathrm{m}\} &= x \Rightarrow (\mathrm{m}\ x) + (\mathrm{m}\ x) \\ h &= x \Rightarrow g\{f\{x \Rightarrow x * x\}\} \end{aligned}$$

Here h calls combinator g, while the macro parameter is also a combinator invocation: $f\{x \Rightarrow x * x\}$. If the outer-most combinator invocation is transformed first, we get:

$$\begin{aligned} h &= x \Rightarrow g'\ x \\ g' &= x \Rightarrow (f\{x \Rightarrow x * x\}\ x) + (f\{x \Rightarrow x * x\}\ x) \end{aligned}$$

Now the $f\{x \Rightarrow x * x\}$ was duplicated in the new function g'. To transform combinator invocations in g', two new functions f' and f'' have to be generated though they are identical:

$$\begin{aligned} h &= x \Rightarrow g'\ x \\ g' &= x \Rightarrow (f'\ x) + (f''\ x) \\ f' &= x \Rightarrow x + (\{x \Rightarrow x * x\}\ x) \\ f'' &= x \Rightarrow x + (\{x \Rightarrow x * x\}\ x) \end{aligned}$$

Though it is possible to avoid generating identical new functions by comparing newly generated functions with previous ones, it would be computationally expensive. How-

ever, if the inner-most combinator invocation is transformed first, we get:

$$\begin{aligned} h &= x \Rightarrow g\{f'\}\ x \\ f' &= x \Rightarrow x + (\{x \Rightarrow x * x\}\ x) \end{aligned}$$

Then the $g\{f'\}$ in h is transformed:

$$\begin{aligned} h &= x \Rightarrow g'\ x \\ g' &= x \Rightarrow (f'\ x) + (f'\ x) \\ f' &= x \Rightarrow x + (\{x \Rightarrow x * x\}\ x) \end{aligned}$$

Only f' is generated by this transformation order. Thus, by transforming inner-most combinator invocations first, we avoid duplicating combinator invocations and creating multiple copies of identical new functions.

However, the above approach does not catch everything. Consider the following program:

$$\begin{aligned} f &= g\{y \Rightarrow y + y\} \\ g\{\mathrm{m}\} &= h\{x \Rightarrow \mathrm{m}\ \mathrm{m}\ x\} \\ h\{\mathrm{m}\} &= k\{x \Rightarrow \mathrm{m}\ \mathrm{m}\ x\} \\ k\{\mathrm{m}\} &= x \Rightarrow \mathrm{m}\ \mathrm{m}\ x \end{aligned}$$

To transform combinator invocations in f, a new function g' is generated:

$$\begin{aligned} f &= g' \\ g' &= h\{x \Rightarrow \{y \Rightarrow y + y\}\ \{y \Rightarrow y + y\}\ x\}. \end{aligned}$$

To transform combinator invocations in g', a new function h' is generated:

$$
\begin{aligned}
f &= g' \\
g' &= h' \\
h' &= k\left\{\begin{array}{ll} x \Rightarrow & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\} \\ & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\}\ x \end{array}\right\}
\end{aligned}
$$

Finally, to transform combinator invocations in h', a new function k' is generated:

$$
\begin{aligned}
f &= g' \\
g' &= h' \\
h' &= k' \\
k' &= \left\{\begin{array}{l} x \Rightarrow \left\{\begin{array}{ll} x \Rightarrow & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\} \\ & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\}\ x \end{array}\right\} \\ \\ \qquad\quad \left\{\begin{array}{ll} x \Rightarrow & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\} \\ & \{x \Rightarrow \{y \Rightarrow y+y\}\ \{y \Rightarrow y+y\}\ x\}\ x \end{array}\right\}\ x \end{array}\right\}
\end{aligned}
$$

Note how the size of the new functions have increased exponentially in each step, this is an example of code explosion discussed earlier. To avoid code explosion, we create a new function for each term that was passed as a macro parameter, if the corresponding macro occurs more than once in the function definition. Taking the above program as example, to transform combinator invocations in f, a new function

p_1 is generated in addition to the new function g':

$$\begin{aligned} f &= g' \\ g' &= h\{x \Rightarrow p_1\, p_1\, x\} \\ p_1 &= y \Rightarrow y + y \end{aligned}$$

Similarly, new functions p_2 and p_3 are generated along h' and k', and the final result is

$$\begin{aligned} f &= g' \\ g' &= h' \\ h' &= k' \\ k' &= x \Rightarrow p_3\, p_3\, x \\ p_3 &= x \Rightarrow p_2\, p_2\, x \\ p_2 &= x \Rightarrow p_1\, p_1\, x \\ p_1 &= y \Rightarrow y + y \end{aligned}$$

Compared with the previous method, this method generates three new functions, but the total size of these functions is much smaller. Similar to the techniques discusses in last section, if the new functions $p_1, \ldots, p_3$ contains environment variables, we can λ-lift them by adding the environment variables as new parameters. And we also need to 'wrap' the call to the new function with an abstraction to make the type of the new abstraction match the type of the old abstraction.

Formally, given $f\{\tau_1, \ldots, \tau_n\}$, where f is a function defined as $v_1, \ldots, v_m \Rightarrow \tau$, we define the following H rule:

$$H[\![f\{\tau_1, \ldots, \tau_n\}]\!] = f\{\tau'_1, \ldots, \tau'_n\}$$

where

$$\tau_i' = \begin{cases} h & \text{occurs}(\mathrm{m}_i, \tau) > 1, \text{env}\langle\tau_i\rangle = \phi \\ w_1, \ldots, w_k \Rightarrow h\ (w_1, \ldots, w_k, e_1, \ldots, e_l) & \text{occurs}(\mathrm{m}_i, \tau) > 1, \text{env}\langle\tau_i\rangle = \{e_1, \ldots, e_l\} \\ \tau_i & \text{occurs}(\mathrm{m}_i, \tau) \leq 1 \end{cases}$$

here occurs(m_i, τ) is the number of times m_i occurs in τ, h is a new function defined as $h = \text{lift}\langle\tau_i\rangle$.

We have changed the M rules described in last section so that it applies the H rule before applying the G rule:

$$M[\![f\{\tau_1, \ldots, \tau_n\}]\!] = G[\![H[\![f\{M[\![\tau_1]\!], \ldots, M[\![\tau_n]\!]\}]\!]]\!]$$

3.4 Simplification and Optimization

The system-generated *fold_D*, *unfold_D* and *map_D* combinators may contain redundant terms, and the procedure to transform combinator invocations may also introduce some unnecessary abstractions. The following rules can be used to simplify and optimize a term:

- Applying the identity function on a term will always generate the same term.

$$\mathrm{O}[\![I\ \tau]\!] = \tau$$

- An abstraction without parameters is equivalent to the term at the right side of the arrow.

$$\mathrm{O}[\![\Rightarrow \tau]\!] = \tau$$

- If an argument to an abstraction is a constant or variable, or the corresponding parameter occurred no more than once in the abstraction, then the parameter can be substituted with the argument.

$$\begin{aligned} & \mathrm{O}[\![\{v_1, \ldots, v_i, \ldots, v_n \Rightarrow \eta\}\ (\tau_1, \ldots, \tau_i, \ldots, \tau_n)]\!] \\ = \ & \mathrm{O}[\![\{v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \Rightarrow \eta[\tau_i/v_i]\}\ (\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n)]\!] \\ & (\text{if } \tau_i \text{ is a constant or a variable, or } \tau_i \text{ occurred no more than once in } \eta) \end{aligned}$$

For example:

$$\mathrm{O}[\![\{x \Rightarrow f\ x\}\ \tau]\!] = \mathrm{O}[\![f\ \tau]\!]$$

Here x occurred only once in the abstraction so it can be replaced by the actual arguments. As another example:

$$\mathrm{O}[\![\{x, y \Rightarrow x * x + y * y\}\ (\tau, 5)]\!] = \mathrm{O}[\![\{x \Rightarrow x * x + 5 * 5\}\ \tau]\!]$$

The parameter 5 corresponding to y is a constant, so the y in the abstraction can be replaced by 5.

- Two abstractions can be combined under certain circumstances:

$$\begin{aligned} & \mathrm{O}[\![\{v_1, \ldots, v_n \Rightarrow \eta\}\ \{w_1, \ldots, w_m \Rightarrow (\tau_1, \ldots, \tau_n)\}\ \tau]\!] \\ = \ & \mathrm{O}[\![\{w_1, \ldots, w_m \Rightarrow \eta[\tau_i/v_i]_{i=1}^{n}\}\ \tau]\!] \end{aligned}$$

If for all i, $1 \leq i \leq n$, τ_i is either a constant or variable, or v_i occurs no more than once in η, then the two abstractions can be combined into one. For example:

$$\mathrm{O}[\![\{x, y \Rightarrow x + y\}\ \{z \Rightarrow (z, z)\}\ \tau]\!] = \mathrm{O}[\![\{z \Rightarrow z + z\}\ \tau]\!]$$

- If a *case* is applied right after the construction of an inductive data, the *case* and the construction can be eliminated:

$$\mathrm{O}[\![\mathrm{case}^D\{\tau_1, \ldots, \tau_n\}\ \ (\mathrm{c}_i^D\ \eta)]\!] = \mathrm{O}[\![\tau_i\ \eta]\!]$$

For example:

$$\mathrm{O}[\![\mathrm{case}^{list}\{\tau_1, \tau_2\}\ \ cons(5, x)]\!] = \mathrm{O}[\![\tau_2\ (5, x)]\!]$$

- If a destructor is applied right after the creation of a record, the destructor and the record creation can be eliminated:

$$\mathrm{O}[\![\mathrm{d}_i\ \mathrm{rec}^D\{\tau_1, \ldots, \tau_n\})]\!] \quad = \quad \mathrm{O}[\![\tau_i]\!]$$

$$\mathrm{O}[\![\mathrm{d}_i^D\ (\eta_1, \ldots, \eta_m, \mathrm{rec}^D\{\tau_1, \ldots, \tau_n\})]\!] \quad = \quad \mathrm{O}[\![\tau_i\ (\eta_1, \ldots, \eta_m)]\!]$$

3.5 Polymorphic functions and basic values

A polymorphic function is a function whose type contains type variables. For example

$$f = x \Rightarrow (x, x)$$

is a polymorphic function whose type is $A \rightarrow A * A$. A is a type variable and can be assigned to any actual type, including basic types such as `int`. However, for efficiency purpose, VMC has two stacks, a basic stack and a pointer stack. The input parameter resides on B-stack if it is a basic value such as an `int`, while it will reside on P-stack if it is a pointer value such as a `list`. The VMC code for these two cases are very different. If we can generate a 'basic' version and a 'pointer' version of f, we can solve this problem. That is, we consider f as a template which has two translations into VMC code: f^B and f^P. f^B is the version to handle basic types and f^P is the version for pointer types. When f is called with actual parameter, the compiler can determine, based on the type of the parameter, which version of f should be used, and generate the corresponding code. For example, consider the function

$$g = () \Rightarrow f\ 3$$

Here f is applied to the argument 3, which has a basic type, so the compiler knows that it must use the code f^B. Unfortunately, there are cases where it is impossible to tell the exact type of an argument, for example:

```
def app = L1,L2 => {| nil: => L2
                    |cons: a,l=> cons(a,l)
                    |} L2.

def start = app([],[]).
```

Here app is of type $list(A), list(A) \Rightarrow list(A)$, it has one type variable A, so two versions of app exist. But in `start`, `[]` could be either a list of basic data or a list of pointer data. Hence there is a choice between version of app. Actually it doesn't matter; both versions will run correctly. In such cases, we will simply call

the version that has already been used somewhere else, to reduce the size of code. If both versions have been used, then the compiler will choose the pointer version.

Multiple versions of constructors also have to be generated to store data more efficiently. Consider the datatype `list`:

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

This datatype *list* has one type variable so that it can represent a list of any datatype, including a list of integers. Figure 3.2 shows two possible ways of storing a list of integers `[5,6]`: Obviously, format (A) is more space efficient. To achieve this, the

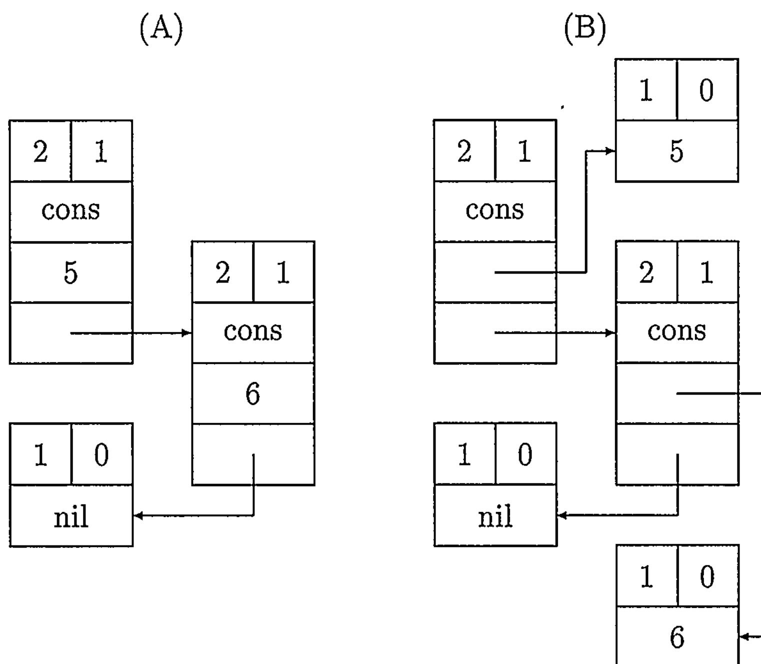


Figure 3.2: Two possible storage formats of [5,6]

constructor cons must have a basic version and a pointer version as well.

The downside of this template-like approach is increased code size. The number of VMC code versions required for a function is exponential in the number of type

variables. In practice, most functions have only a small number of type variables so this worst case scenario does not happen. However, to cater for those cases in which the number of type variables is large, the compiler generates the different versions by demand only.

A lot of research has been done on the use of basic (unboxed) values with polymorphic functions. One commonly used method is to box the basic values before passing them to the polymorphic function and unbox the result when necessary [23, 31]. But the cost of boxing/unboxing can be very high for complicated data structures. For example, the type of

$$p0 = x \Rightarrow \{(y0, y1) \Rightarrow y0\}\ x$$

is $A * B \to A$. When passing a pair of integers to $p0$, we have to project the components of the tuple, box them, then build a new tuple. The cost of boxing/unboxing outweighs the cost of the polymorphic function. An alternative approach, by using Intensional Type Analysis, is to pass types as arguments to polymorphic functions in order to determine the representation of an object [16, 34]. This approach avoids the cost of boxing/unboxing, at the price of run time type analysis. The template-like approach is more efficient and has been used, for instance, by Blelloch in a NESL compiler [9], and by Jones to eliminate Haskell overloading [21]. Furthermore, Jones reports that this approach does not lead to excessive code-blowup.

3.6 Type Checking Again

The first type checking ensures the validity of the program. However, to generate VMC code, we also need type information:

- In order to decide which version of a polymorphic function is being called, we need the exact type of that call.
- To decide which stack a variable resides on, we need the type of the variable.
- For an abstraction, the type of the result is needed so that we know which stack the result resides on.

For these reasons, a second pass of type-checking is done in order to gather the needed type information. We do this in three phases: first, we add type tags (actually type variable identifiers) to every tuple, abstraction, and function call. Then we decompose the term and generate type equations. Last, we solve the equations and now we can determine the actual types of each tagged terms. The following ML datatype is used to denote the tagged terms.

```
datatype Tagged
   = tINT of integer                  (* integer constant *)
   | tSTR of string                   (* string constant *)
   | tVAR of string                   (* variable *)
   | tTUPLE of Tagged list* int list (* n-tuple, including 0-tuple*)
   | tAPP of Tagged * Tagged          (* application *)
   | tRECORD of string * Tagged list (* record *)
   | tCASE of string * Tagged list   (* case *)
   | tCALL of string * int list      (* call a function *)
   | tABS of (string * int) list * Tagged * int (* abstraction *)
```

The datatype `Tagged` is similar to the `Term` defined before, except at this stage, all the combinator invocations have been transformed into normal function calls. All the type tags are type-variables, so we actually use an integer to denote the type variable. However, for clarity, we will represent them as a type expression here. The type tags for a function call (including constructors and destructors) corresponds to

the type variables in the type of the function. For example, a call to function f of type $list(A), list(B) \rightarrow A * B$, will be tagged with a list of new type variables $[X, Y]$, which corresponds to A and B. If X is solved to be of type int (a basic type), and Y to be of type $string$ (a pointer type), then we know the version we are calling is f^{BP}. The type tags for an abstraction are type tags corresponding to each of the variables declared in the abstraction and a type variable corresponds to the result type. We will represent a tagged term by τ^t, where τ is the term and t is the type variable.

Translation from a term to a tagged term is straightforward, we simply go over the whole term and tag terms with a new type variables. For every function call f, we look up the symbol table and find its type, if there are m type variables in its type, we tag f with $[X_1, \ldots, X_m]$, where X_i is a new type variable.

The following rules decompose a tagged term and generate type equations (on the right):

- Integer constant:

$$\frac{\Gamma \vdash i : T}{.} \quad int = T$$

- String constant:

$$\frac{\Gamma \vdash s : T}{.} \quad string = T$$

- Variable:

$$\frac{\Gamma, x : S \vdash x : T}{.} \quad S = T \qquad \frac{\Gamma, y : S \vdash x : T}{\Gamma \vdash x : T} \quad (x \neq y)$$

- Null-tuple:

$$\frac{\Gamma \vdash () : T}{\cdot} \quad 1 = T$$

- n-Tuple:

$$\frac{\Gamma \vdash (\tau_1, \ldots, \tau_n)^{X_1,\ldots,X_n} : T}{\Gamma \vdash \tau_1 : X_1 \quad \cdots \quad \tau_n : X_n} \quad X_1 * \cdots * X_n = T$$

- Application:

$$\frac{\Gamma \vdash (\tau_1\ \tau_2) : T}{\Gamma \vdash s : A \quad \Gamma \vdash f : A \rightarrow T}$$

- Abstraction: here $X_1, \ldots, X_n \Rightarrow Y$ is the type tag associated with the abstraction.

$$\frac{\Gamma \vdash v_1, \ldots, v_n \Rightarrow \tau^{X_1,\ldots,X_n \rightarrow Y} : S \rightarrow T}{\Gamma, v_1 : X_1, \ldots, v_n : X_n \vdash t : Y} \quad Y = T, X_1 * \cdots * X_n = S$$

- Function call: here X_i are the type variables tagged on f, the type of f (by looking up the symbol table) is in the form $E_1, \ldots, E_k \rightarrow F$, type variables within E_i are $T_1, \ldots, T_n$.

$$\frac{\Gamma \vdash f^{X_1,\ldots,X_n} : S \rightarrow T}{\cdot} \quad (E_1 * \cdots * E_k)[X_i/T_i]_{i=1}^{n} = S, F = T$$

- Record: here coinductive datatype D is defined as

$$\text{data } C \to D(T_1, \ldots, T_m) = \left| \begin{array}{lcl} d_1 & : & H_1^1, H_1^2, \ldots, H_1^{k_1}, C \to E_1 \\ d_2 & : & H_2^1, H_2^2, \ldots, H_2^{k_2}, C \to E_2 \\ \vdots & & \\ d_n & : & H_n^1, H_n^2, \ldots, H_n^{k_n}, C \to E_n \end{array} \right.$$

Then the rule is

$$\frac{\Gamma \vdash \text{rec}^L\{\tau_1, \ldots, \tau_n\} : T}{\Gamma \vdash \tau_1 : F_1 \quad \cdots \quad \Gamma \vdash \tau_n : F_n} \quad L(A_1, \ldots, A_m) = T$$

where $F_i = (H_i^1 * H_i^2 * \cdots * H_i^{k_i} \to E_i)[L(A_1, \ldots, A_m)/C][A_i/T_i]_{i=1}^m$

- Case: here D is defined as

$$\text{data } D(T_1, \ldots, T_m) \to C = \left| \begin{array}{lcl} c_1 & : & E_1^1, E_1^2, \ldots, E_1^{k_1} \to C \\ c_2 & : & E_2^1, E_2^2, \ldots, E_2^{k_2} \to C \\ \vdots & & \\ c_n & : & E_n^1, E_n^2, \ldots, E_n^{k_n} \to C \end{array} \right.$$

Then the rule is

$$\frac{\Gamma \vdash \text{case}^D\{\tau_1, \ldots, \tau_n\} : S \to T}{\Gamma \vdash \tau_1 : F_1 \to T \quad \cdots \quad \Gamma \vdash \tau_n : F_n \to T} \quad D(A_1, \ldots, A_m) = S$$

where $F_i = (E_i^1 * E_i^2 * \cdots * E_i^{k_i})[L(A_1, \ldots, A_m)/C][A_i/T_i]_{i=1}^m$

The *unification* [33] algorithm is then applied on the set of equations generated: a solution is always obtained as the validity of the program has been guaranteed by the

first type-checking pass. Then the term is traversed and the solution is substituted into the type variables which tag the terms. Finally, as we are only concerned with whether the tag is a basic type or pointer type, we re-tag a term τ as τ^B if the term is a basic type, or as τ^P if it is a pointer type.

`INT` i	$\rightarrow i$
`STR` s	$\rightarrow s$
`VAR` "x"	$\rightarrow x$
`TUPLE` $[\tau_1, \ldots, \tau_n]$	$\rightarrow (\tau_1, \ldots, \tau_n)$
`APP`(τ_1, τ_2)	$\rightarrow \tau_1\ \tau_2$
`RECORD`$(D, [\tau_1, \ldots, \tau_n])$	$\rightarrow \texttt{rec}^D\{\tau_1, \ldots, \tau_n\}$
`CASE`$(D, [\tau_1, \ldots, \tau_n])$	$\rightarrow \texttt{case}^D\{\tau_1, \ldots, \tau_n\}$
`MACRO` i	$\rightarrow \texttt{m}_i$
`CTR(D,i)`	$\rightarrow \texttt{c}_i^D$
`DTR(D,i)`	$\rightarrow \texttt{d}_i^D$
`CALL`("f", $[\tau_1, \ldots, \tau_n]$)	$\rightarrow f\{\tau_1, \ldots, \tau_n\}$
`ABS`$([x_1, \ldots, x_n], \tau)$	$\rightarrow x_1, \ldots, x_n \Rightarrow \tau$

Table 3.1: Representation of ML datatype `Term`

Chapter 4

VMC Code Generation

4.1 Overview

After a Charity program is translated into VMC code, its variables during execution reside on stacks and are referenced by their positions relative to the top of the stacks. Since values are frequently pushed onto or popped off stacks, the position of a variable is hard to calculate. To solve this problem, we simulate the stacks by using a list K to represent current status of the stacks. Each element in K is of the form of v^X, where v is the name of a variable (An empty string ϕ is used if it is an intermediate value which has no name). X can be either B or P, representing the type of the variable. The list K allows us to determine which stack a variable resides on and its current position.

The following subsidiary functions are used in code generation:

- $\mathit{type}(K, v)$ searches for v in K, starting at the head, and returns the type tag of v, which is either B or P.

- $\mathit{pos}(K, v)$ searches for v in K, starting at the head and ending at v, then returns the number of elements encountered which is of the same type as v. For example, if v is of type B, then it returns the number of basic elements before v.

- $\mathit{bcnt}(S)$ counts the number of B elements in S, which is a set of type tags.

- $pcnt(S)$ counts the number of P elements in S.

4.2 Code Generation Rules

Given a term tagged with types as described in last chapter, we generate a list of VMC code using the following three sets of rules. The F rule is used on the top-level abstraction of a function only. The R rule is used on the phrases of a record, and the T rules are used for all the other circumstances. Each rule generates a VMC code list as the result. The R rules may generate a separate code list, which are not immediately concatenated to the other code lists.

- Top-level abstraction: each function body is an abstraction, it needs to be treated differently because there is a return address on the B-stack and we also need to generate the return instruction. Here $cb = bcnt(\{X_1, \ldots, X_n\})$ and $cp = pcnt(\{X_1, \ldots, X_n\})$

$$F[\![\{v_1, \ldots, v_n \Rightarrow \tau\}^{X_1,\ldots,X_n \to B}]\!] = \boxed{\begin{array}{l} T[\![[\phi^B, v_n^{X_n}, \ldots, v_1^{X_1}], \tau]\!] \\ \texttt{retB}\ cb\ cp \end{array}}$$

$$F[\![\{v_1, \ldots, v_n \Rightarrow \tau\}^{X_1,\ldots,X_n \to P}]\!] = \boxed{\begin{array}{l} T[\![[\phi^B, v_n^{X_n}, \ldots, v_1^{X_1}], \tau]\!] \\ \texttt{retP}\ cb\ cp \end{array}}$$

- Abstraction: we need to pop off the input arguments and move the results to their proper positions. If the abstraction has more than one argument, we *detuple* first, because in this case, its argument is a tuple on P-stack. A sim-

ple optimization will eliminate any consecutive *newtuple* and *detuple* (section 4.3.4). $v :: K$ denotes a list with v as its head and K as its tail. Code inside [] is generated only if the condition specified at the subscript to the brackets is satisfied.

$$T[\![K, \{v_1, \ldots, v_n \Rightarrow \tau\}^{X_1,\ldots,X_n \to Y}]\!] = \boxed{\begin{array}{l} [\ \texttt{detuple}\ ]_{n>1} \\ T[\![v_n^{X_n} :: \cdots :: v_1^{X_1} :: K, \tau]\!] \\ [\ \texttt{moveB}\ \ 0\ \ cb\ ]_{Y=B} \\ [\ \texttt{moveP}\ \ 0\ \ cp\ ]_{Y=P} \\ \texttt{popB}\ \ cb \\ \texttt{popP}\ \ cp \end{array}}$$

where

$$\begin{aligned} cb &= bcnt(\{X_1, \ldots, X_n\}) \\ cp &= pcnt(\{X_1, \ldots, X_n\}) \end{aligned}$$

- Integer constant:

$$T[\![K, i]\!] = \boxed{\texttt{constI}\ \ i}$$

- String constant:

$$T[\![K, s]\!] = \boxed{\texttt{newstr}\ \ s}$$

- Variable:

$$T[\![K, v]\!] = \begin{cases} \boxed{\texttt{dupB}\ \ pos(K, v)} & \text{if } type(K, v) = B \\ \\ \boxed{\texttt{dupP}\ \ pos(K, v)} & \text{if } type(K, v) = P \end{cases}$$

- n-Tuple: here the empty string ϕ stands for an intermediate value that has no name.

$$T[\![K, (\tau_1, \ldots, \tau_n)^{X_1,\ldots,X_n}]\!] = \boxed{\begin{array}{l} T[\![K, \tau_1]\!] \\ T[\![\phi^{X_1} :: K, \tau_2]\!] \\ \vdots \\ T[\![\phi^{X_{n-1}} :: \phi^{X_{n-2}} :: \cdots :: \phi^{X_1} :: K, \tau_n]\!] \\ \texttt{newtuple}\ \ cb\ \ cp \end{array}}$$

- Application:

$$T[\![K, \tau_1\ \tau_2]\!] = \boxed{\begin{array}{l} T[\![K, \tau_2]\!] \\ T[\![K, \tau_1]\!] \end{array}}$$

- Function call: Let m be the number of arguments of f, n be the number of type variables of f and $f^{X_1 X_2 \ldots X_n}$ a version of f with its type variables equal

to $X_1, \ldots, X_n$.

$$T[\![K, f^{X_1,X_2,\ldots,X_n}]\!] = \begin{cases} \boxed{\begin{array}{l}\texttt{detuple} \\ \texttt{call}\ \ f^{X_1X_2\ldots X_n}\end{array}} & \text{if } m > 1 \\ \boxed{\texttt{call}\ \ f^{X_1X_2\ldots X_n}} & \text{if } m \leq 1 \end{cases}$$

- Built-in function call: (e.g. the operator +) the body of these functions are pre-defined lists of VMC code.

$$T[\![K, f]\!] = \begin{cases} \boxed{\texttt{body of}\ f} & \text{if } f \text{ has 0 or 1 argument} \\ \boxed{\begin{array}{l}\texttt{detuple} \\ \texttt{body of}\ f\end{array}} & \text{if } f \text{ has more than one arguments} \end{cases}$$

- Constructor application: $c_i^{X_1,\ldots,X_n}$ is a version of the constructor with its type

variables equal to $X_1, \ldots, X_n$. Denote the type of $c_i^{X_1,\ldots,X_n}$ as $E_1, \ldots, E_m \to F$.

$T[\![K, c_i^{X_1,\ldots,X_n}]\!] = \begin{cases} \boxed{\texttt{detuple} \\ \texttt{constI } i \\ \texttt{newtuple } cb{+}1 \ cp} & \text{if } m > 1 \\ \boxed{\texttt{constI } i \\ \texttt{newtuple } cb{+}1 \ cp} & \text{if } m <= 1 \end{cases}$

where

$$cb = bcnt(E_1, \ldots, E_m)$$
$$cp = bcnt(E_1, \ldots, E_m)$$

- Destructor application:

$T[\![K, d_i]\!] = \begin{cases} \boxed{\texttt{detuple} \\ \texttt{getfieldP } i \\ \texttt{detuple} \\ \texttt{xcall}} & \text{if } d_i \text{ is higher-order} \\ \boxed{\texttt{dupP 0} \\ \texttt{getfieldP } i \\ \texttt{detuple} \\ \texttt{xcall}} & \text{if } d_i \text{ is not higher-order} \end{cases}$

- Case: here L_i and L_E are unique labels.

$$T[\![K, \text{case}^D\{\tau_1, \ldots, \tau_n\}]\!] = \boxed{\begin{array}{ll} & \texttt{detuple} \\ & \texttt{case}\ L_1 \ldots L_n \\ L_1: & \\ & T[\![K, \tau_1]\!] \\ & \texttt{goto}\ L_E \\ L_2: & \\ & T[\![K, \tau_2]\!] \\ & \texttt{goto}\ L_E \\ L_3: & \\ & \vdots \\ L_n: & \\ & T[\![K, \tau_n]\!] \\ L_E: & \end{array}}$$

- Record: a *record* is represented in the VMC as a tuple of closures. Each closure is composed of a code pointer (which is actually a integer) and its parameters. First, we introduce an auxiliary rule P, which generates code to push a list of

variables onto the stack. The X_i below is the type tag of v_i.

$$P[\![K, [v_1, v_2, \ldots, v_n]]\!] = \boxed{\begin{array}{l} T[\![K, v_1]\!] \\ T[\![v_1^{X_1} :: K, v_2]\!] \\ T[\![v_2^{X_2} :: v_1^{X_1} :: K, v_3]\!] \\ \vdots \\ T[\![v_{n-1}^{X_{n-1}} :: \cdots :: v_1^{X_1} :: K, v_n]\!] \end{array}}$$

where $X_i = \mathit{type}(K, v_i)$.

We now give the rule to translate *record*:

$$T[\![K, \mathrm{rec}^D\{\tau_1, \ldots, \tau_n\}]\!]$$

$$= \boxed{\begin{array}{l} P[\![K, \mathrm{env}\langle\tau_1\rangle]\!] \\ \texttt{constA}\ L_1 \\ \texttt{newtuple}\ b_1{+}1\ p_1 \\ P[\![\phi^P :: K, \mathrm{env}\langle\tau_2\rangle]\!] \\ \texttt{constA}\ L_2 \\ \texttt{newtuple}\ b_2{+}1\ p_2 \\ P[\![\phi^P :: \phi^P :: K, \mathrm{env}\langle\tau_3\rangle]\!] \\ \vdots \\ P[\![\phi^P :: \cdots :: \phi^P :: K, \mathrm{env}\langle\tau_n\rangle]\!] \\ \texttt{newtuple}\ 0\ n \end{array}} \quad \text{and} \quad \boxed{\begin{array}{l} L_1: \\ R[\![1, \tau_1]\!] \\ L_2: \\ R[\![2, \tau_2]\!] \\ \vdots \\ L_n: \\ R[\![n, \tau_n]\!] \end{array}}$$

$\text{env}\langle\tau\rangle$ is the list of environment variables of τ, b_i is the number of basic variables in $\text{env}\langle\tau_i\rangle$, and p_i is the number of pointer variables in $\text{env}\langle\tau_i\rangle$. The code generated for a *record* consists two parts: the first part builds the tuple of closures; the second part is the collection of code segments that are being pointed to by the closures. The second part is a separate piece of code and should not be concatenated with the first part.

For clarity, the second part relies on another rule R. The R rule generates the suspended computation code by first λ-lifting the abstraction (recall: each phrase in a *record* is in the form of an abstraction, even if the corresponding destructor is not higher-order), then translating the lifted abstraction by the F or T rule. The following rule is applied if destructor i is higher-order. $e_1, \ldots, e_k$ are environment variables of $\{v_1, \ldots, v_m \Rightarrow\}$, $Y_j = \mathit{type}(K, e_j)$, $Z = \{X_1, \ldots, X_m, Y_1, \ldots, Y_k\}$.

$$
\begin{aligned}
& R[\![i, \{v_1, \ldots, v_m \Rightarrow t\}^{X_1,\ldots,X_m \rightarrow T}]\!] \\
= \;& F[\![\{v_1, \ldots, v_m, e_1, \ldots, e_k \Rightarrow t\}^{X_1,\ldots,X_m,Y_1,\ldots,Y_k \rightarrow T}]\!]
\end{aligned}
$$

If destructor i is not higher-order, the closure needs to be updated with the

result.

$$R[\![i, \{\Rightarrow \tau\}^{\to B}]\!] = $$

```
        T[[[], {e_1, ..., e_k => τ}^{Y_1,...,Y_k→B}]]
        dupB 0
        constA L_B
        newtuple 2 0
        setfieldP 1 i
L_B:
        retB 0 1
```

$$R[\![i, \{\Rightarrow \tau\}^{\to P}]\!] = $$

```
        T[[[], {e_1, ..., e_k => τ}^{Y_1,...,Y_k→P}]]
        dupP 0
        constA L_P
        newtuple 1 1
        setfieldP 2 i
L_P:
        retP 0 1
```

4.3 Optimizations

Many of the code generation rules can be modified to generate more efficient code. The following optimizations are described in this section:

- Basic optimizations to datatype representation (4.3.1, 4.3.2, 4.3.3).
- Peephole optimization on the VMC code, including the `newtuple`/`detuple`

elimination which is crucial in reducing heap usage.

4.3.1 Simple inductive datatypes

If an inductive datatype has no type argument and none of its constructors has any arguments, we call it a simple inductive datatype. The `boolean` datatype, for example, is a simple inductive datatype:

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

A simple inductive datatype can always be represented by a small integer. If we represent simple inductive datatypes as integers, we can improve speed and memory efficiency because now we do not need to box/unbox it. To achieve this, we treat simple inductive datatypes as basic types in type-checking, and the code generation rules need to be modified:

- The constructors for simple inductive datatypes simply put an integer on top of the B-stack.

$$T[\![K, \mathrm{c}_i]\!] = \boxed{\texttt{constI}\ i}$$

- The *case* on a simple inductive data does not unbox the data:

$$T[\![K, \text{case}^D\{\tau_1,\ldots,\tau_n\}]\!] = \begin{array}{|lll|} & & \texttt{case}\ L_1\ldots L_n \\ L_1: & & \\ & & T[\![K,\tau_1]\!] \\ & & \texttt{goto}\ L_E \\ L_2: & & \\ & & T[\![K,\tau_2]\!] \\ & & \texttt{goto}\ L_E \\ L_3: & & \\ & & \vdots \\ L_n: & & \\ & & T[\![K,\tau_n]\!] \\ L_E: & & \end{array}$$

4.3.2 Inductive datatype with only one constructor

If an inductive datatype has only one constructor, we do not need to store the constructor number in the heap node. Consequently, we need to change the following code generation rules:

Constructor: if the constructor has multiple arguments, we simply create a tuple of its arguments. However, since the arguments are already in a tuple, the constructor actually does nothing! If the constructor has only one argument, we simply box it. $\text{c}^{X_1,X_2,\ldots,X_n}$ is a version of constructor with its type variables

equal to $X_1, \ldots, X_n$. Denote the type of $c^{X_1,\ldots,X_n}$ as $E_1, \ldots, E_m \to F$.

$$T[\![K, c^{X_1,\ldots,X_n}]\!] = \begin{cases} \epsilon & \text{if } m > 1 \\ \boxed{\texttt{newtuple 1 0}} & \text{if } m = 1 \text{ and } E_1 = B \\ \boxed{\texttt{newtuple 0 1}} & \text{if } m = 1 \text{ and } E_1 = P \\ \boxed{\texttt{constI 0}} & \text{if } m = 0 \end{cases}$$

Case: a case on inductive data with only one constructor does not generate the case instruction

$$T[\![K, \text{case}^D\{\tau\}]\!] = \boxed{\begin{array}{l} \texttt{detuple} \\ T[\![K, \tau]\!] \end{array}}$$

4.3.3 Higher-order coinductive datatypes with only one destructor

If a coinductive datatype has only one destructor, and the destructor is higher-order, then we can use a closure itself as the *record* instead of creating a 1-tuple of closures. The following code generation rules are used:

- Record:

$$T[\![K, \text{rec}^D\{\tau\}]\!] = \boxed{\begin{array}{l} P[\![K, \text{env}\langle\tau\rangle]\!] \\ \texttt{constA } L_D \\ \texttt{newtuple } cb{+}1 \ cp \end{array}} + \boxed{\begin{array}{l} L_D : \\ \qquad R[\![K, 1, \tau]\!] \end{array}}$$

where

$$\begin{aligned} cb &= bcnt(\text{env}\langle\tau\rangle) \\ cp &= pcnt(\text{env}\langle\tau\rangle) \end{aligned}$$

- Destructor:

$$T[\![K, \mathrm{d}]\!] = \boxed{\begin{array}{l}\texttt{detuple}\\ \texttt{detuple}\\ \texttt{xcall}\end{array}}$$

This optimization is very useful, because Charity does not support higher-order functions directly, instead, higher-order functions are usually implemented by defining a coinductive datatype such as

$$\text{data } C \to exp(A, B) = fn : C \to A \Rightarrow B.$$

This optimization makes the implementation of higher-order functions as efficient as languages that support higher-order functions directly.

4.3.4 Peephole optimizations on VMC code

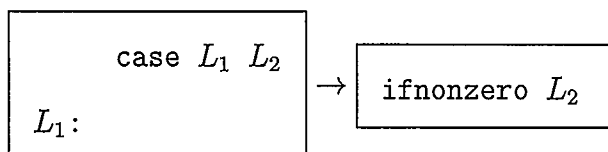
As the last step of translation, some simple peephole optimizations are applied on the generated VMC code.

- If a `newtuple` instruction is immediately followed by a `detuple` instruction, then these two instructions can be eliminated.

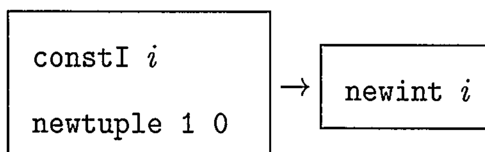
$$\boxed{\begin{array}{l}\texttt{newtuple } m\ n\\ \texttt{detuple}\end{array}} \to \epsilon$$

This optimization is critical in reducing heap usage. For a Charity expression $f(x, y)$, we generate the codes for (x, y) first, then the code for calling f. The last instruction for (x, y) is a `newtuple`, and the first instruction for calling f is a `detuple` if f is a function with multiple arguments. Obviously the `newtuple` and `detuple` are unnecessary. It is possible to change the code generation rules to avoid generating such code, but that makes the rules very complex. Instead, the above optimization at the VMC code level is sufficient.

- A `case` L_1 L_2 immediately followed by the definition of L_1 can be translated into a `ifnonzero` L_2 The latter can usually be implemented more efficiently on modern CPUs.



- A `constI` i immediately followed by a `newtuple 1 0` can be replaced by a `newint` i instruction. A constructor without any arguments (such as the *nil* in the definition of *list*) will be represented by a boxed small integer. Since such constructors will not be updated, the virtual machine creates only one heap node for each small integer and re-use it. Heap usage can be reduced by up to 25 percent by this optimization. Note this optimization requires that the boxed integer will never be updated.



- The following instructions have no effect at all and can be simply eliminated:

`popB 0`	$\rightarrow \epsilon$
`popP 0`	$\rightarrow \epsilon$
`moveB` i i	$\rightarrow \epsilon$
`moveP` i i	$\rightarrow \epsilon$

Chapter 5

Results and Conclusion

This thesis describes the Virtual Machine for Charity and the detailed translation steps from Charity code to VMC code. The compiler is written in SML, and the virtual machine is implemented as an interpreter written in C++.

This new implementation is significantly faster than the old one and consumes much less memory. The following table shows the speed and memory usage of running the same programs under the new implementation and the old one. The test environment is Intel Xeon 1.4G running Window 2000.

Program	Time(new)	Time 1(old)	Time 2(old)	Heap size(new)	Heap size(old)
pegs(1)	0.42s	7s	2s	2M	5M
pegs(2)	7s	163s	21s	32M	80M
primes	7.4s	71s	57s	1M	2.5M

pegs is a program to solve the peg solitaire game. It uses an exhaustive search to find the solution. *pegs(1)* and *pegs(2)* are the same program with different input. *primes* is a program that finds the first 1000 prime numbers.

The old interpreter starts with a very small heap, and increases the heap size gradually as more memory is needed. This accounts for the run-time decrease between the initial execution and subsequent execution of a program in the old interpreter. To be fair, we measured both times. The new implementation does not have this

issue so repeated executions takes the same amount of time. The old interpreter does not provide memory usage statistics, so the heap size of the old implementation is a rough estimate observed from Windows 2000's task manager.

The contributions of this thesis are:

Design and specification of VMC: The structure of VMC is simpler than many other virtual machines, yet it can represent both inductive data and coinductive data efficiently. Basic values (unboxed) can be stored and represented directly in VMC, and a heap node can contain a mixture of basic values and heap pointers.

The translation from Charity code to VMC code and related optimizations: The translation aims to reduce heap usage and to improve execution speed. This is achieved by the following approaches:

1. The new multi-function types (3.1.1) allow several function parameters to be passed on stack, eliminating tuple formation. The multi-function types also allow more efficient representation of data structures.
2. Combinators invocations are transformed into normal functions calls by macro-substitution (3.3), which avoids the cost of building closures for macro parameters.
3. Polymorphic functions are compiled into different versions (3.5) so that basic data can be represented and manipulated in their natural unboxed form. It also allows more efficient representation of polymorphic datatypes.
4. The intermediate representation is simplified and optimized following the

rules in section 3.4.

5. Code generation rules are optimized to allow more efficient representation of certain datatypes (section 4.3.1,4.3.2, 4.3.3).

6. Some peephole optimizations are applied on the generated VMC code (section 4.3.4).

Implementation of the core-Charity translator and the VMC interpreter: The core-Charity translator consists of 2000 lines of SML code, much shorter and cleaner than the previous implementation. The VMC interpreter provides a rich set of commands to debug VMC programs.

Possible future work could be:

- Implementing pattern-matching [32]. This feature is available in old implementation but has not been implemented in the new implementation, due to limited time.

- Implementing new features of Charity, such as modules, more extensive I/O, etc.

- Compiling VMC code into native machine code. VMC is designed to model a concrete machine, so it is relatively easy to map VMC code into native machine code. The main issue here might be how to make use of registers efficiently.

- Proving the correctness of the translation rules. One of the more interesting aspects of Charity is that it is based uncompromisingly on the formal theory of datatypes. Thus, it is highly suitable to implementations for which proofs

of correctness are required. It would therefore be very nice to be assured that the implementation of Charity itself is correct!

Bibliography

[1] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszynński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 1–13, 1991.

[3] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[4] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, February 1989.

[5] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 89–100. MIT Press, Cambridge, Massachusetts, 1991.

[6] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[7] J. Backus. Can programming be liberated from the Von Neumann style? *Communications of the ACM*, 21(8):613–641, 1978.

[8] R. Bird and P. L. Wadler. *Introduction to Functional Programming.* Prentice Hall, Hemel Hempstead, 1988. (pbk).

[9] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[10] Robin Cockett. Charitable thoughts, 1996. (draft lecture notes, `http://www.cpsc.ucalgary.ca/projects/charity/home.html`).

[11] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[12] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 50–64. Springer-Verlag, Berlin, DE, 1985.

[13] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 34–46. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[14] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[15] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986.

[16] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[17] Mike Hermann. A lazy graph reduction machine for Charity: CHarity Abstract Reduction Machine (CHARM). (unfinished), July 1992.

[18] P. R. Hudak, P. L. Wadler, et al. Report on the programming language haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[19] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[20] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture*, New York, NY, USA, 1985. Springer-Verlag Inc.

[21] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[22] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[23] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[24] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[25] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[26] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Reading, MA, 1993.

[27] P. M. Sansom and S. L. Peyton Jones. Generational garbage collection for Haskell. Research Report FP-1993-2, Department of Computer Science, University of Glasgow, Glasgow, UK, 1993.

[28] Marc A. Schroeder. Higher-order Charity. Master's thesis, The University of Calgary, July 1997.

[29] Sjaak Smesters, Erick Nöcker, Jon van Gronigen, and Rinus Plasmeijer. Generating efficient code for lazy functional languages. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 592–617. Springer Verlag, June 1991.

[30] Dwight L. Spencer. *Categorical Programming with Functorial Strength.* PhD thesis, The Oregon Graduate Institute of Science and Technology, January 1993.

[31] Peter Thiemann. Unboxed values and polymorphic typing revisited. In *Functional Programming Languages and Computer Architecture*, pages 24–35, 1995.

[32] Charles Tuckey. Pattern matching in Charity. Master's thesis, The University of Calgary, July 1997.

[33] Peter Vesely. Typechecking the Charity term logic, April 1997. (documentation, `http://www.cpsc.ucalgary.ca/projects/charity/home.html`).

[34] Stephanie Weirich. Encoding intensional type analysis. *Lecture Notes in Computer Science*, 2028, 2001.

[35] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[36] Dale Barry Yee. Implementing the Charity abstract machine. Master's thesis, The University of Calgary, September 1995.

Appendix A

VMC specification

A.1 Formal Syntax of VMC program

VMC Program	→	code_list
code_list	→	code | code separator code_list
separator	→	`LF` | `BAR`
code	→	instruction | label : | ! identifier
instruction	→	operator | operator operands
operator	→	identifier
operands	→	operand | operand operands
operand	→	label | number | string
label	→	`[@|_] [a-z|A-Z|0-9|_]*`
identifier	→	`[a-z|A-Z] [a-z|A-Z|0-9|_]*`

A.2 Data

A.2.1 Basic data

Char	8-bit signed character.
Short	16-bit signed integer.
Integer	32-bit signed integer.
Long	64-bit signed integer.
Float	32-bit real number.
Double	64-bit real number
Address	Internally a 32-bit integer, but represented by labels in VMC code.

A.2.2 Pointer data

tuple	Each field must be 32-bits, but can be of different type.
array	Each element must be of the same type, but not restricted to 32-bit data.
string	An array of Char with 0 at the end of the string.

A.3 Instruction Set

Instructions are specified by state transition tables. The state of VMC can be described by four elements: Instruction pointer, B-stack, P-stack, Heap. Elements not affected by the instruction are not shown in the tables. Symbols used in the transition tables are:

$\{instr\}.cs$	IP status. *instr* is the next instruction to be executed. *cs* represent the code sequence following *instr*.
$b.bs$	B-stack status. b is the top element bs represents all the elements under b.
$p.ps$	P-stack status. p is the top element ps represents all the elements under p.
b	any 32-bit basic value
u	any 64-bit basic value
p, q	pointer
$\|cs\|$	the address of code sequence cs
c	Char (8-bit)
s	Short (16-bit)
i	Integer (32-bit)
f	Float (32-bit)
l	Long (64-bit)
d	Double (64-bit)
$(0 : c)$	32 bit data, the lowest 8 bits be c and other bits be 0
$(0 : s)$	32 bit data, the lowest 16 bits be s and other bits be 0
$(b_1 \cdots b_m p_1 \cdots p_n)$	a tuple with m basic values and n pointers.
$[b_1 b_2 \ldots b_n]$	an array with n items.
k, j, m, n	non-negative integer.

A.3.1 Arithmetic instructions

All Arithmetic instructions take input from the B-stack, pop off the inputs and push the result on B-stack. The last character in the instruction name indicates the type of operands: I for Integer, F for Float, L for Long and D for Double.

`addI,addF,addL,addD`	addition
`subI,subF,subX,subD`	subtraction
`mulI,mulF,mulL,mulD`	multiplication
`divI,divF,divL,divD`	division
`modI,modF,modL,modD`	modulus
`negI,negF,negL,negD`	negation

Code	B-stack	$\Rightarrow$	Code	B-stack	Comment
{addI}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 + i_2$
{addF}.cs	$f_2.f_1.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = f_1 + f_2$
{addL}.cs	$l_2.l_1.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = l_1 + l_2$
{addD}.cs	$d_2.d_1.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = d_1 + d_2$
{subI}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 - i_2$
{subF}.cs	$f_2.f_1.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = f_1 - f_2$
{subL}.cs	$l_2.l_1.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = l_1 - l_2$
{subD}.cs	$d_2.d_1.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = d_1 - d_2$
{mulI}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 \times i_2$
{mulF}.cs	$f_2.f_1.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = f_1 \times f_2$
{mulL}.cs	$l_2.l_1.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = l_1 \times l_2$
{mulD}.cs	$d_2.d_1.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = d_1 \times d_2$
{divI}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 \div i_2$
{divF}.cs	$f_2.f_1.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = f_1 / f_2$
{divL}.cs	$l_2.l_1.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = l_1 \div l_2$
{divD}.cs	$d_2.d_1.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = d_1 / d_2$
{modI}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 \bmod i_2$
{modF}.cs	$f_2.f_1.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = f_1 \bmod f_2$
{modL}.cs	$l_2.l_1.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = l_1 \bmod l_2$
{modD}.cs	$d_2.d_1.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = d_1 \bmod d_2$
{negI}.cs	$i.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = -i$
{negF}.cs	$f.bs$	$\Rightarrow$	cs	$f'.bs$	$f' = -f$
{negL}.cs	$l.bs$	$\Rightarrow$	cs	$l'.bs$	$l' = -l$
{negD}.cs	$d.bs$	$\Rightarrow$	cs	$d'.bs$	$d' = -d$

A.3.2 Logical and bit instructions

Logical and bit instructions take inputs from B-stack, pop off the inputs and then put the result on the B-stack. All instructions operate on integers only.

`and,or,not`	logical And, Or, Not
`bitand,bitor,bitnot`	bitwise And, Or, Not
`bitxor`	bitwise eXclusive Or
`shl,shr`	shift left, shift right

Code	B-stack	$\Rightarrow$	Code	B-stack	Comment
{`and`}.cs	$0.0.bs$	$\Rightarrow$	cs	$0.bs$	
{`and`}.cs	$0.i.bs$	$\Rightarrow$	cs	$0.bs$	$i \neq 0$
{`and`}.cs	$i.0.bs$	$\Rightarrow$	cs	$0.bs$	$i \neq 0$
{`and`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$1.bs$	$i_1 \neq 0, i_2 \neq 0$
{`or`}.cs	$0.0.bs$	$\Rightarrow$	cs	$0.bs$	
{`or`}.cs	$0.i.bs$	$\Rightarrow$	cs	$1.bs$	$i \neq 0$
{`or`}.cs	$i.0.bs$	$\Rightarrow$	cs	$1.bs$	$i \neq 0$
{`or`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$1.bs$	$i_1 \neq 0, i_2 \neq 0$
{`not`}.cs	$i.bs$	$\Rightarrow$	cs	$0.bs$	$i \neq 0$
{`not`}.cs	$0.bs$	$\Rightarrow$	cs	$1.bs$	
{`bitand`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1$ bitand i_2
{`bitor`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1$ bitor i_2
{`bitnot`}.cs	$i.bs$	$\Rightarrow$	cs	$i'.bs$	$i' =$ bitnot i
{`bitxor`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1$ bitxor i_2
{`shl`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 \times 2^{i_2}$
{`shr`}.cs	$i_2.i_1.bs$	$\Rightarrow$	cs	$i'.bs$	$i' = i_1 \div 2^{i_2}$

A.3.3 Comparison instructions

Except for the eqP instruction, all comparison instructions take inputs from B-stack, pop off the inputs and put the result on B-stack. The result is always a 0 or 1. The last character of each instruction indicate the type of operands: I for Integer, F for Float, L for Long and D for Double.

`eqI,eqF,eqL,eqD` equal

`gtI,gtF,gtL,gtD` greater than

`geI,geF,geL,geD` greater than or equal to

`eqP` test whether two pointers are equal

Code	B-stack	P-stack	$\Rightarrow$	Code	B-stack	P-stack	Condition
{`eqI`}.cs	$i.i.bs$		$\Rightarrow$	cs	$1.bs$		
{`eqI`}.cs	$i.i'.bs$		$\Rightarrow$	cs	$0.bs$		$i \neq i'$
{`eqF`}.cs	$f.f.bs$		$\Rightarrow$	cs	$1.bs$		
{`eqF`}.cs	$f.f'.bs$		$\Rightarrow$	cs	$0.bs$		$f \neq f'$
{`eqL`}.cs	$l.l.bs$		$\Rightarrow$	cs	$1.bs$		
{`eqL`}.cs	$l.l'.bs$		$\Rightarrow$	cs	$0.bs$		$l \neq l'$
{`eqD`}.cs	$d.d.bs$		$\Rightarrow$	cs	$1.bs$		
{`eqD`}.cs	$d.d'.bs$		$\Rightarrow$	cs	$0.bs$		$d \neq d'$
{`eqP`}.cs	bs	$p.p.ps$	$\Rightarrow$	cs	$1.bs$	ps	
{`eqP`}.cs	bs	$p.q.ps$	$\Rightarrow$	cs	$0.bs$	ps	$p \neq q$
{`gtI`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$1.bs$		$i_1 > i_2$
{`gtI`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$0.bs$		$i_1 \leq i_2$
{`gtF`}.cs	$f_2.f_1.bs$		$\Rightarrow$	cs	$1.bs$		$f_1 > f_2$
{`gtF`}.cs	$f_2.f_1.bs$		$\Rightarrow$	cs	$0.bs$		$f_1 \leq f_2$
{`gtL`}.cs	$l_2.l_1.bs$		$\Rightarrow$	cs	$1.bs$		$l_1 > l_2$
{`gtL`}.cs	$l_2.l_1.bs$		$\Rightarrow$	cs	$0.bs$		$l_1 \leq l_2$
{`gtD`}.cs	$d_2.d_1.bs$		$\Rightarrow$	cs	$1.bs$		$d_1 > d_2$
{`gtD`}.cs	$d_2.d_1.bs$		$\Rightarrow$	cs	$0.bs$		$d_1 \leq d_2$
{`geI`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$1.bs$		$i_1 \geq i_2$
{`geI`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$0.bs$		$i_1 < i_2$
{`geF`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$1.bs$		$f_1 \geq f_2$
{`geF`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$0.bs$		$f_1 < f_2$
{`geL`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$1.bs$		$l_1 \geq l_2$
{`geL`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$0.bs$		$l_1 < l_2$
{`geD`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$1.bs$		$d_1 \geq d_2$
{`geD`}.cs	$i_2.i_1.bs$		$\Rightarrow$	cs	$0.bs$		$d_1 < d_2$

A.3.4 Data conversion instructions

Data conversion instructions convert one basic datum to the equivalent basic datum of another type. Each instruction takes one input from the B-stack, pops off the input and puts the result on the B-stack. The first letter of each instruction name indicates the type of input, the last letter of each instruction name indicates the type of output: c for Char, s for Short, i for Integer, f for Float, l for Long and d for Double.

`c2i`	Char to Integer
`s2i`	Short to Integer
`i2c,i2s,i2f,i2l,i2d`	Integer to Char, Short, Float, Long and Double
`f2i,f2l,f2d`	Float to Integer, Long and Double
`l2i,l2f,l2d`	Long to Integer, Float and Double
`d2i,d2f,d2l`	Double to Integer, Float and Long

Code	B-stack	$\Rightarrow$	Code	B-stack	Comment
{c2i}.cs	$(0:c).bs$	$\Rightarrow$	cs	$i.bs$	$i=c$
{s2i}.cs	$(0:s).bs$	$\Rightarrow$	cs	$i.bs$	$i=s$
{i2c}.cs	$i.bs$	$\Rightarrow$	cs	$(0:c).bs$	$c=i$
{i2s}.cs	$i.bs$	$\Rightarrow$	cs	$(0:s).bs$	$s=i$
{i2f}.cs	$i.bs$	$\Rightarrow$	cs	$f.bs$	$f=i$
{i2l}.cs	$i.bs$	$\Rightarrow$	cs	$l.bs$	$l=i$
{i2d}.cs	$i.bs$	$\Rightarrow$	cs	$d.bs$	$d=i$
{f2i}.cs	$f.bs$	$\Rightarrow$	cs	$i.bs$	$i=\lfloor f\rfloor$
{f2l}.cs	$f.bs$	$\Rightarrow$	cs	$l.bs$	$l=\lfloor f\rfloor$
{f2d}.cs	$f.bs$	$\Rightarrow$	cs	$d.bs$	$d=f$
{l2i}.cs	$l.bs$	$\Rightarrow$	cs	$i.bs$	$i=l$
{l2f}.cs	$l.bs$	$\Rightarrow$	cs	$f.bs$	$f=l$
{l2d}.cs	$l.bs$	$\Rightarrow$	cs	$d.bs$	$d=l$
{d2i}.cs	$d.bs$	$\Rightarrow$	cs	$i.bs$	$i=\lfloor d\rfloor$
{d2l}.cs	$d.bs$	$\Rightarrow$	cs	$l.bs$	$l=\lfloor d\rfloor$
{d2f}.cs	$d.bs$	$\Rightarrow$	cs	$f.bs$	$f=d$

A.3.5 Branch instructions

Unconditional branch instructions either go to a constant address specified in the instruction or take the address from the B-stack. Conditional branch instructions check the top value on the B-stack to determine whether to go to a specific address. `retX` instructions pops off the inputs for subroutine and keeps the result on top of stacks.

`goto` L	goto a specific address L
`ret`	pops off the top value on the B-stack then goto there
`call` L	push current address on the B-stack then goto L
`xcall`	same as `call` except the goto address is on top of B-stack
`case` $L_0 \cdots L_n$	goto L_i where i is the top value on B-stack
`ifzero` L	goto L if the top value of the B-stack is zero
`ifnonzero` L	goto L if the top value of the B-stack is not zero
`ifpos` L	goto L if the top value of the B-stack is positive
`ifneg` L	goto L if the top value of the B-stack is negative
`retM` m n i j	pop off i basic values and j pointers as the result, then pop off one basic value as the return address, then pop off m basic values and n pointers, push the result back and go to the return address.
`retB` m n	equivalent to `retM` m n `1 0`
`retP` m n	equivalent to `retM` m n `0 1`
`retU` m n	equivalent to `retM` m n `2 0`
`halt`	stop the execution of virtual machine.

Code	B-stack	$\Rightarrow$	Code	B-stack	Condition
$\{\texttt{goto }\vert\texttt{cs'}\vert\}.cs$		$\Rightarrow$	cs'		
$\{\texttt{ret}\}.cs$	$\vert cs'\vert.bs$	$\Rightarrow$	cs'	bs	
$\{\texttt{call }\vert\texttt{cs'}\vert\}.cs$	bs	$\Rightarrow$	cs'	$\vert cs\vert.bs$	
$\{\texttt{xcall}\}.cs$	$\vert cs'\vert.bs$	$\Rightarrow$	cs'	$\vert cs\vert.bs$	
$\{\texttt{case }\vert cs'_0\vert\cdots\vert cs'_k\vert\}.cs$	$i.bs$	$\Rightarrow$	cs'_i	bs	$0 \leq i \leq k$
$\{\texttt{ifzero }\vert\texttt{cs'}\vert\}.cs$	$0.bs$	$\Rightarrow$	cs'	bs	
$\{\texttt{ifzero }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs	bs	$i \neq 0$
$\{\texttt{ifnonzero }\vert\texttt{cs'}\vert\}.cs$	$0.bs$	$\Rightarrow$	cs	bs	
$\{\texttt{ifnonzero }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs'	bs	$i \neq 0$
$\{\texttt{ifpos }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs'	bs	$i > 0$
$\{\texttt{ifpos }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs	bs	$i \leq 0$
$\{\texttt{ifneg }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs'	bs	$i < 0$
$\{\texttt{ifneg }\vert\texttt{cs'}\vert\}.cs$	$i.bs$	$\Rightarrow$	cs	bs	$i \geq 0$
$\{\texttt{halt}\}.cs$		$\Rightarrow$	ϵ		

	Code	B-stack	P-stack
	$\{\texttt{retM } m\ n\ i\ j\}.cs$	$b'_1.\cdots.b'_i.\vert cs'\vert.b_1.\cdots.b_m.bs$	$p'_1.\cdots.p'_j.p_1.\cdots.p_n.p$
$\Rightarrow$	cs'	$b'_1.\cdots.b'_i.bs$	$p'_1.\cdots.p'_j.ps$
	$\{\texttt{retB } m\ n\}.cs$	$b'.\vert cs'\vert.b_1.\cdots.b_m.bs$	$p_1.\cdots.p_n.ps$
$\Rightarrow$	cs'	$b'.bs$	ps
	$\{\texttt{retP } m\ n\}.cs$	$\vert cs'\vert.b_1.\cdots.b_m.bs$	$p'.p_1.\cdots.p_n.p$
$\Rightarrow$	cs'	bs	$p'.ps$
	$\{\texttt{retU } m\ n\}.cs$	$b'_1.b'_2.\vert cs'\vert.b_1.\cdots.b_m.bs$	$p_1.\cdots.p_n.ps$
$\Rightarrow$	cs'	$b'_1.b'_2.bs$	ps

A.3.6 Constant instructions

Constant instructions put a constant on top of B-stack. The last character of the instruction name indicate the type of the constant: I for Integer, F for Float, L for Long and D for double, A for address (label).

`constI` i	push a Integer on top of B-stack
`constF` f	push a Float on top of B-stack
`constL` l	push a Long on top of B-stack
`constD` d	push a Double on top of B-stack
`constA` L	push address of label L on top of B-stack

Code	B-stack	$\Rightarrow$	Code	B-stack
{`constI` i}.cs	bs	$\Rightarrow$	cs	$i.bs$
{`constF` f}.cs	bs	$\Rightarrow$	cs	$f.bs$
{`constL` l}.cs	bs	$\Rightarrow$	cs	$l.bs$
{`constD` d}.cs	bs	$\Rightarrow$	cs	$d.bs$
{`constA` $\|cs'\|$}.cs	bs	$\Rightarrow$	cs	$\|cs'\|.bs$

A.3.7 Stack instructions

Stack instructions manipulate items on B-stack and P-stack. The last character in the instruction name indicate which stack to manipulate and the size of data: B for B-stack 32 bit item, U for B-stack 64 bit item and P for P-stack.

`dupB, dupU, dupP`	duplicate an item and put it on top of stack
`moveB, moveU, moveP`	set the value of one item to the value of another
`swapB, swapU, swapP`	swap the top two item

	Code	B-stack	P-stack
	{dupB k}.cs	$b_0.b_1.\cdots.b_k.bs$	
$\Rightarrow$	cs	$b_k.b_0.b_1.\cdots.b_k.bs$	
	{dupU k}.cs	$b_0.b_1.\cdots.b_k.b_{k+1}.s$	
$\Rightarrow$	cs	$b_k.b_{k+1}.b_0.b_1.\cdots.b_k.b_{k+1}.bs$	
	{dupP k}.cs		$p_0.p_1.\cdots.p_k.ps$
$\Rightarrow$	cs		$p_i.p_0.p_1.\cdots.p_k.ps$
	{popB n}.cs	$b_1.\cdots.b_n.bs$	
$\Rightarrow$	cs	bs	
	{popP n}.cs		$p_1.\cdots.p_n.ps$
$\Rightarrow$	cs		ps
	{moveB j k}.cs	$b_0.\cdots.b_{k-1}.b_k.b_{k+1}.\cdots.b_j.bs$	
$\Rightarrow$	cs	$b_0.\cdots.b_{k-1}.b_j.b_{k+1}.\cdots.b_j.bs$	
	{moveB j k}.cs	$b_0.\cdots.b_j.\cdots.b_{k-1}.b_k.bs$	
$\Rightarrow$	cs	$b_0.\cdots.b_j.\cdots.b_{k-1}.b_j.bs$	
	{moveU j k}.cs	$b_0.\cdots.b_{k-1}.b_k.b_{k+1}.\cdots.b_j.b_{j+1}.bs$	
$\Rightarrow$	cs	$b_0.\cdots.b_{k-1}.b_j.b_{j+1}.\cdots.b_j.b_{j+1}.bs$	
	{moveU j k}.cs	$b_0.\cdots.b_j.b_{j+1}.\cdots.b_{k-1}.b_k.b_{k+1}.bs$	
$\Rightarrow$	cs	$b_0.\cdots.b_j.b_{j+1}.\cdots.b_{k-1}.b_j.b_{j+1}.bs$	
	{moveP j k}.cs		$p_0.\cdots.p_{k-1}.p_k.p_{k+1}.\cdots.p_j.ps$
$\Rightarrow$	cs		$p_0.\cdots.p_{k-1}.p_j.p_{k+1}.\cdots.p_j.ps$
	{moveP j k}.cs		$p_0.\cdots.p_j.\cdots.p_{k-1}.p_k.ps$
$\Rightarrow$	cs		$p_0.\cdots.p_j.\cdots.p_{k-1}.p_j.ps$
	{swapB}.cs	$b_1.b_2.bs$	
$\Rightarrow$	cs	$b_2.b_1.bs$	
	{swapU}.cs	$u_1.u_2.bs$	
$\Rightarrow$	cs	$u_2.u_1.bs$	
	{swapP}.cs		$p_1.p_2.ps$
$\Rightarrow$	cs		$p_2.p_1.ps$

A.3.8 Tuple instructions

Tuple instruction create and manipulate tuples. The last capitalized character indicates the type of the value in the tuple: B for basic value, P for heap pointer, and U for 64 bit basic value. The `newint` instruction is for an optimization purpose: A constructor without argument (such as the *nil* in the definition of *list*) will be represented by a boxed small integer. Since such constructor will not be updated, the virtual machine can create only one tuple for each small integer and re-use them. Heap usage is reduced significantly by this optimization.

`newtuple` m n	create a tuple with m basic data and n pointer data on stacks.
`detuple`	load all fields of a tuple onto stacks.
`getfieldB` i	put the value of field i on B-stack
`getfieldP` i	put the value of field i on P-stack
`getfieldU` i	put the value of 64 bit field i on B-stack
`setfieldB` i	set the value of field i to the top 32 bit value on B-stack
`setfieldP` i	set the value of field i to the top 64 bit value on B-stack
`setfieldU` i	set the value of field i to the top pointer on P-stack
`tuplesizeB`	get the number of basic fields in a tuple
`tuplesizeP`	get the number of pointer fields in a tuple
`newint` i	create a 1-tuple which contains an constant integer i.

	Code	B-stack	P-stack	Heap
	{`newtuple` m n}.cs	$b_1.\cdots.b_m.bs$	$p_1.\cdots.p_n.ps$	
$\Rightarrow$	cs	b	$q.ps$	$q \rightarrow (b_1 \cdots b_m p_1 \cdots p_n)$
	{`detuple`}.cs	bs	$q.ps$	$q \rightarrow (b_1 \cdots b_m p_1 \cdots p_n)$
$\Rightarrow$	cs	$b_1.\cdots.b_m.bs$	$p_1.\cdots.p_n.ps$	
	{`getfieldB` k}.cs	bs	$q.ps$	$q \rightarrow (x_0 \cdots x_k \cdots)$
$\Rightarrow$	cs	$x_k.bs$	ps	
	{`getfieldU` k}.cs	bs	$q.ps$	$q \rightarrow (x_0 \cdots x_k x_{k+1} \cdots)$
$\Rightarrow$	cs	$x_k.x_{k+1}.bs$	ps	
	{`getfieldP` k}.cs		$q.ps$	$q \rightarrow (x_0 \cdots x_k \cdots)$
$\Rightarrow$	cs		$x_k.ps$	
	{`setfieldB` k}.cs	$b'.bs$	$q.ps$	$q \rightarrow (x_0 \cdots x_k \cdots)$
$\Rightarrow$	cs	bs	ps	$q \rightarrow (x_0 \cdots b' \cdots)$
	{`setfieldU` k}.cs	$b_l.b_r.bs$	$q.ps$	$q \rightarrow (x_0 \cdots x_k x_{k+1} \cdots)$
$\Rightarrow$	cs	bs	ps	$q \rightarrow (x_0 \cdots b_l b_r \cdots)$
	{`setfieldP` k}.cs		$p'.q.ps$	$q \rightarrow (x_0 \cdots x_k \cdots)$
$\Rightarrow$	cs		ps	$q \rightarrow (x_0 \cdots p' \cdots)$
	{`tuplesizeB`}.cs	bs	$q.ps$	$q \rightarrow (b_1 \cdots b_m p_1 \cdots p_n)$
$\Rightarrow$	cs	$m.bs$	ps	
	{`tuplesizeP`}.cs	bs	$q.ps$	$q \rightarrow (b_1 \cdots b_m p_1 \cdots p_n)$
$\Rightarrow$	cs	$n.bs$	ps	
	{`newint` i}.cs	bs	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow (i)$

A.3.9 Array instructions

Array instructions create and manipulate arrays. The capitalized last character in the instruction name indicate the type of array items: C for Char, S for Short, B for 32-bit basic data, U for 64-bit basic data and P for pointer.

`newArray`X	create a array, each item is of type X
`getitem`X	get the value of an item in an array, the item is of type X
`setitem`X	set the value of an item in an array, the item is of type X
`arraysize`X	get the size of an array, item in the array is of type X

	Code	B-stack	P-stack	Heap
	{newarrayC}.cs	$n.bs$	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [c_0 c_1 \cdots c_{n-1}]$
	{newarrayS}.cs	$n.bs$	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [s_0 s_1 \cdots s_{n-1}]$
	{newarrayB}.cs	$n.bs$	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [b_0 b_1 \cdots b_{n-1}]$
	{newarrayU}.cs	$n.bs$	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [u_0 u_1 \cdots u_{n-1}]$
	{newarrayP}.cs	$n.bs$	ps	
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [p_0 p_1 \cdots p_{n-1}]$
	{getitemC}.cs	$k.bs$	$q.ps$	$q \rightarrow [c_0 c_1 \cdots c_{n-1}]$
$\Rightarrow$	cs	$c_k.bs$	ps	
	{getitemS}.cs	$k.bs$	$q.ps$	$q \rightarrow [s_0 s_1 \cdots s_{n-1}]$
$\Rightarrow$	cs	$(0 : s_k).bs$	ps	
	{getitemB}.cs	$k.bs$	$q.ps$	$q \rightarrow [b_0 b_1 \cdots b_{n-1}]$
$\Rightarrow$	cs	$b_k.bs$	ps	
	{getitemU}.cs	$k.bs$	$q.ps$	$q \rightarrow [u_0 u_1 \cdots u_{n-1}]$
$\Rightarrow$	cs	$u_k.bs$	ps	
	{getitemP}.cs	$k.bs$	$q.ps$	$q \rightarrow [p_0 p_1 \cdots p_{n-1}]$
$\Rightarrow$	cs	bs	$p_i.ps$	
	{setitemC}.cs	$(0 : c').k.bs$	$q.ps$	$q \rightarrow [c_0 \cdots c_{k-1} c_k c_{k+1} \cdots c_{n-1}]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [c_0 \cdots c_{k-1} c' c_{k+1} \cdots c_{n-1}]$
	{setitemS}.cs	$(0 : s').k.bs$	$q.ps$	$q \rightarrow [s_0 \cdots s_{k-1} s_k s_{k+1} \cdots s_{n-1}]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [s_0 \cdots s_{k-1} s' s_{k+1} \cdots s_{n-1}]$
	{setitemB}.cs	$b'.k.bs$	$q.ps$	$q \rightarrow [b_0 \cdots b_{k-1} b_k b_{k+1} \cdots b_{n-1}]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [b_0 \cdots b_{k-1} b' b_{i+1} \cdots b_{n-1}]$
	{setitemU}.cs	$u'.k.bs$	$q.ps$	$q \rightarrow [u_0 \cdots u_{k-1} u_k u_{k+1} \cdots u_{n-1}]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [u_0 \cdots u_{k-1} u' u_{k+1} \cdots u_{n-1}]$
	{setitemP}.cs	$k.bs$	$p'.q.ps$	$q \rightarrow [p_0 \cdots p_{k-1} p_k p_{k+1} \cdots p_{n-1}]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [p_0 \cdots p_{k-1} p' p_{k+1} \cdots p_{n-1}]$
	{arraysizeC}.cs	bs	$q.ps$	$q \rightarrow [c_0 c_1 \cdots c_{n-1}]$
$\Rightarrow$	cs	$n.bs$	ps	
	{arraysizeS}.cs	bs	$q.ps$	$q \rightarrow [s_0 s_1 \cdots s_{n-1}]$
$\Rightarrow$	cs	$n.bs$	ps	
	{arraysizeB}.cs	bs	$q.ps$	$q \rightarrow [b_0 b_1 \cdots b_{n-1}]$
$\Rightarrow$	cs	$n.bs$	ps	
	{arraysizeU}.cs	bs	$q.ps$	$q \rightarrow [u_0 u_1 \cdots u_{n-1}]$
$\Rightarrow$	cs	$n.bs$	ps	
	{arraysizeP}.cs	bs	$q.ps$	$q \rightarrow [p_0 p_1 \cdots p_{n-1}]$
$\Rightarrow$	cs	$n.bs$	ps	

A.3.10 String instructions

String instructions create and manipulate strings. A string is stored in a character array, with a null character in the array indicating the end of string, equivalent to a string in the C programming language. So array instructions can be used to get or set a single character of the string.

`newstr` *str*	create an array which contains *str* and a trailing null character
`strlen`	get the length of a string, excluding the trailing null character.
`strcat`	create a new string by concatenating two strings

	Code	B-stack	P-stack	Heap
	{`newstr` "$c_0 \cdots c_n$"}.cs		ps	
$\Rightarrow$	cs		$q.ps$	$q \rightarrow [c_0 c_1 \cdots c_n 0]$
	{`strlen`}.cs	bs	$q.ps$	$q \rightarrow [c_1 \cdots c_n 0]$
$\Rightarrow$	cs	$n.bs$	ps	
	{`strcat`}.cs	bs	$p'.p.ps$	$p \rightarrow [c_1 \cdots c_n 0], p' \rightarrow [c'_1 \cdots c'_n 0]$
$\Rightarrow$	cs	bs	$q.ps$	$q \rightarrow [c_1 \cdots c_n c'_1 \cdots c'_n 0]$

Appendix B

VMC interpreter user's manual

The VMC interpreter is an implementation of VMC according to the VMC specification. An interpreter will not provide optimal performance, but it makes testing and debugging VMC code easier.

B.1 Configuration

Two versions of the VMC interpreter exist, a normal version and a 'fast' version. The normal version does a lot of validity checks at run time and will not crash if the VMC code is incorrect. For example, before the execution of a `popP` n instruction, the interpreter checks if there are at least n items on the P-stack. If the check fails, the interpreter stops the execution of the program and prints an error message. However, these integrity checks slow the machine down significantly. The fast version assumes the correctness for VMC code it executes and does not do these checks. The fast version should be used to run VMC code that is known to be valid, for example, the code generated by the Charity compiler.

Before loading VMC code into memory, the interpreter calls the C preprocessor (cpp) to process the VMC code, so all the preprocessor commands in C (e.g. `#include`, `#define`, etc.) can be used in VMC code. This makes manually writing VMC code easier.

The interpreter contains three files, all of which should be put into the same

directory and the directory needs to be in the search path for executables.

`vmc` The normal version of the VMC interpreter

`vmcfast` The 'fast' version of the VMC interpreter

`cpp` The C preprocessor, which is used by the interpreter to preprocess VMC code before loading.

B.2 Command line syntax

The command line syntax of the interpreter is as following, where `vmcfast` replaces `vmc` if the fast version is desired.

```
vmc [-?] [-Heap:m] [-Stack:n] [filename]
```

The `-?` parameter request help, the interpreter will display information of the command line syntax.

The `-Heap:m` is the optional parameter to specify the size of the heap in VMC, in megabytes. The default value of `m` is 8.

The `-Stack:n` is the optional parameter to specify the size of the stacks in VMC, in megabytes. The B-stack and P-stack in the interpreter share the same piece of memory and grow in opposite directions. So this parameter specifies the total size of the stacks. The default value of `n` is 1 megabyte.

The `filename` is optional. If no filename is given, the interpreter enters interactive mode, which will be discussed in next section. Otherwise, the interpreter enters execution mode and executes the specified file until it halts. If the filename has no suffix, then `.vmc` is appended as its suffix.

Here is an example that run a file `test.vmc`, with heap size set to 16MB and stack size set to 4MB.

```
vmc -Heap:16 -Stack:4 test
```

B.3 Commands

Run the interpreter without given a filename, the interpreter enters interactive mode, and shows the following message:

```
Virtual Machine for Charity version 1.02
copyright (c) 2001,2002 Charity Development Group
Enter a command, type help for a list of available commands
VMC>>
```

Now the interpreter is ready to accept user commands. To see the list of available commands, you may use the `help` command.

```
VMC>> help

load filename -- load a program into memory, default suffix is .vmc
reload        -- reset the system then load the last file
exec filename -- load & run a program
run  [address]-- run from the first instruction or a given address
thread n -- change current thread to thread n
stat [n] -- show status of thread n or current thread
clear    -- clear the stacks and the heap
reset    -- restore to initial states
quit     -- exit the program
ip  [address] -- show or set current address
d   address   -- display tuple at heap address
dstr address  -- display string at heap address
d?  address   -- display array, ? is array type which can be c,s,i,l,f,d
u   [address] -- unassembly
p             -- proceed(step over)
```

```
t              -- trace(step through)
g   [address] -- run from current address and stop at given address
setbp address -- set break point
clrbp address -- clear break point
clrbp          -- clear all break points
showbp         -- show current break points
istk  n        -- show the nth element on B-stack, as an int
fstk  n        -- show the nth element on B-stack, as an float
lstk  n        -- show the nth element on B-stack, as an long
dstk  n        -- show the nth element on B-stack, as an double
pstk  n        -- show the nth element on P-stack

An address is either a number or a label
VMC>>
```

Here is a explanation for each command:

load The `load` *filename* command pre-processes a source file by calling cpp, then loads the result into the code store. If the filename has no extension, the default extension `.vmc` is assumed. If a program is composed of several source files, they can be loaded one by one. However, if file A references a label in file B, then file B must be loaded before file A.

reload This command has no parameters. It re-initializes the system first, then load the most recently loaded file. If you are debugging a VMC program and make some modifications to the source code, this command can be very convenient.

exec This command is equivalent to a `load` command followed by a `run` command.

thread VMC supports multiple threads, each thread has its own IP and stacks. Most of the commands here operate on the current thread only. This command sets the current thread.

stat This command shows the status of current thread. Here is an example status

```
VMC>> stat
Thread 0  HEAP USED:  64 Unit GC=0
STACK USED: 5 / 262144
B-Stack(3): 10, 20, 9, #
P-Stack(2): 35, 44{2}, #
VMC>>
```

The first line indicates current thread number, how much heap space has been used (each unit is 32-bit) and how many times garbage collection has been run. The second line shows how much stack space has been used and the total size of stack space. Each stack unit is 32 bit. The third line shows the top 10 items on the B-stack, as integers, starting from the top item. If there are less than 10 items, # is shown indicating the bottom of the stack. The fourth line shows the top 10 items on the P-stack. Here pointers are represented as integers. The `d` command is used to view the heap item referenced by a pointer. If a pointer points to a 1-element boxed integer, that integer is directly shown inside the curly brackets. So in the above example, 44{2} means a pointer which points to a boxed integer 2.

clear Clear the stacks and heap, but keep the loaded programs and breakpoints.

reset re-initialize the interpreter, stacks, heap, breakpoints and clear all loaded programs. This is equivalent to restarting the interpreter.

quit exit the interpreter. You may also press ctrl-d (under unix) or ctrl-z (under windows) to quit the interpreter.

ip This command has an optional parameter, which can be either a label or a number. When the optional parameter is present, this command sets IP (the instruction pointer) to the specified address. Otherwise it just shows current IP.

d display the heap data referenced by a pointer.

dc,ds,di,df,dl,dd display the heap data as an array of Char, Short, Integer, Float, Long or Double

u Unassemble from current IP or a specified address.

p Single step execution without entering procedure calls. If current instruction is a `call` or `xcall`, stop only after the subroutine returns.

t Single step execution with entering procedure calls. This command differs from p in that it stop immediately after execute one instruction, no matter what the instruction is.

g There is one optional parameter for this command, which specifies the stop address. This command will execute from current IP till the specified stop address, a breakpoint or `halt` is encountered.

setbp Set a breakpoint. The break point can be represented either by a number or a label.

clrbp Clear a specific breakpoint if an address is given as parameter, else clear all the breakpoints.

showbp Show all the breakpoints.

istk Show an item on the B-stack as an integer. The argument of this command specifies the position of item, with the top item being in position 0.

fstk,lstk,dstk Similar to `istk` except showing the item as Float, Long, or Double correspondingly.

pstk Show an item on the P-stack. The argument of this command specifies the position of the item, with the top item being in position 0.