The Implementation Of Pattern Matching In Charity

Final Project Report For CPSC 502

Charles Tuckey
Department of Computer Science
University of Calgary
CANADA
email: tuckey@cpsc.ucalgary.ca

April 4, 1994

Abstract

Charity is a functional type language recently developed at the University of Calgary by Dr. J. R. B. Cockett. This paper is a report on a project to add pattern matching capability to **charity**. The project involved defining an extended term logic for **charity**, developing an appropriate syntax to express the new term logic, doing type checking in the term logic, and refining an algorithm to translate the extended term logic to the core term logic. The preceding work is discussed and a blueprint is given for future work.

Acknowledgements

The ideas in this project come from several sources. The definition of complete patterns came from Dr. Robin Cockett. Tom Fukushima's term logic type checking program was the basis for the algorithm used here; although it had to be modified somewhat. The pattern matching term logic, the syntax for pattern matched terms, and the translation algorithm were developed by the author with assistance from the above individuals.

In addition, both Robin Cockett and Tom Fukushima read drafts of this report and offered much helpful commentary. Thanks also to the other members of the **charity** group: Marc Schroeder, Peter Vesely and Barry Yee.

Thanks to Dr. Mike Williams for doing his best to keep things on track and moving from a project management point of view.

In spite of all the above help, errors and omissions have certainly crept into this report. The author takes full responsibility for those.

The author was supported in part by an NSERC Undergraduate Research Award from May to August 1993. It was during this time that he was first exposed to **charity** and pattern matching.

Preface

A reader familiar with **charity** will find the going much easier in this report than a reader unfamiliar with **charity** . However, chapters 1 and 2 are meant to serve as a brief introduction to the **charity** programming language. Chapter 1 gives a very brief overview of functional languages in general. Chapter 2 gives some examples of programs written in **charity** and shows how the major characteristics of functional languages are exemplified in the language.

Chapter 3 provides some motivation for developing pattern matching, although any programmer familiar with functional languages will find this unnecessary. Chapter 4 discusses various types of patterns and how they relate to **charity** and Miranda.

The details involved in implementing the project are given in chapter 5.

In chapter 6 patterns in **charity** are precisely defined. Also, an extended term logic is defined and a programming syntax for expressing pattern matching is developed.

Chapter 7 lays out the term logic type checking algorithm and chapter 8 gives an algorithm to translate the extended term logic to core term logic.

Plans for future work are given in chapter 9.

Appendix A contains code written in core term logic for a parser. It is intended to serve as motivation for pattern matching.

Appendices B and C contain variable bases and terms from the core term logic that are replaced in the extended term logic.

Appendices D and E contain grammars for the core term logic and extended term logic respectively.

Charity is available by ftp from `ftp.cpsc.ucalgary.ca` in `:pub/projects/charity`. Of course, this is only the core term logic version but the pattern matching version will be available soon!

Contents

List of Figures

Chapter 1

Functional Languages Overview

The class of computer programming languages can be broken into two major subclasses: procedural languages such as C and assembly level languages; and declarative languages. Declarative languages can be further subdivided into three groups (Holyer, 1991):

- logical programming languages,
- functional languages, and
- specification languages.

Procedural languages are designed to let the programmer work directly with the machine by modifying the values in memory locations. Steps are performed *sequentially*; each step modifying the state of the machine.

Declarative languages, on the other hand, let the programmer work directly with values by specifying relations and functions between values without regard to how they are stored.

Logic programming languages, such as Prolog, use relations to express relationships between values. Functional languages, such as SML and Miranda[1], use functions. Specification languages, such as Z, are not really programming languages as such, but rather give the user a way to give precise specifications about tasks to be carried out. These can then be checked for soundness and correctness. Specfication languages offer both relations and functions.

Charity is a functional language. The next section discusses some of the characteristics of functional languages in general. Chapter 2 describes **charity** and how it exemplifies these

[1]Miranda is a trademark of Research Software Ltd.

characteristics.

1.1 Functional Languages

The basic building blocks of functional languages are functions. Functions are combined to form other functions; in fact, a complete functional program *is* a function. A mathematical function provides exactly one output for any given input and will always produce the same output given the same input. A function in a functional language behaves in exactly the same way. Therefore, we can say that a function will produce no side effects and has no concept of the state of the machine outside of its input arguments. This is the major difference between functions in a functional language and procedures in procedural languages.

The above described characteristics of functional languages give rise to the following list of advantages often cited for functional languages as compared to procedural languages (Hughes, 1989; Backus, 1978):

- rapid development of programs,
- ease of maintenance and enhancement,
- semantic simplicity,
- mathematically tractable,
- easier to do programming proofs and transformations, and
- suitable for use on a parallel computer.

Most modern functional languages share a common set of attributes which include:

- datatypes,
- static typing,
- polymorphism,
- pattern matching,
- higher order functions, and
- lazy evaluation.

Every piece of data in a functional program has a type much as in procedural languages such as Pascal. *Static typing* means the types cannot change during program execution. Therefore, typechecking can be performed on a program at compile time to ensure that functions are being called with data of the proper type.

Unlike Pascal, however, many functional languages have a *polymorphic* type system that allows functions to take on a variety of types under different circumstances.

Often functions can be treated in functional programs as data. *Higher order functions* are functions that take functions as input or return functions as a result.

It is not always the case that all possible values of a data structure need to be evaluated in order to produce a result. *Lazy evaluation* is used to have the program compute only what is needed to generate a function's output.

Pattern matching, the subject of this paper, provides the programmer with a shorthand way of describing the action a function should take given a certain kind of input.

The next chapter describes how the above attributes are exemplified in the functional language **charity**.

Chapter 2

Charity

Charity , a language developed recently at the University of Calgary by Dr. Robin Cockett, belongs to the class of functional languages and thus retains the usual advantages cited for functional languages. But **charity** is novel among functional languages because its mathematical base is category theory and not, as is generally true of other functional languages, the lambda calculus. The ties between **charity** and its mathematical parent are very close. This has resulted in a programming language which moves easily from mathematical program specifications to program code, has distinct advantages when it comes to producing verified programs and is guaranteed to terminate(Cockett & Fukushima, 1992).

2.1 Category Theory and Charity

Charity is based on category theory. A category can be thought of as an algebra of functions with composition as the main operation. A category **A** is formally defined as (Walters, 1991):

a set of *objects*, obj $\mathbf{A}$, and a set of *morphisms* such that:

1. Each morphism has a designated *domain* and *codomain* in obj $\mathbf{A}$. When the domain of f is A and the codomain of f is B we write $f : A \longrightarrow B$.

2. Given morphisms $f : A \longrightarrow B, g : B \longrightarrow C$, there is a designated *composite* morphism

$$f; g : A \longrightarrow C.$$

3. Given any object A there is a designated *identity* morphism

$$1_A : A \longrightarrow A.$$

4. The data above is required to satisfy the following:

   - Identity laws. If $f : A \longrightarrow B$ then

   $$f; 1_B = f \text{ and } 1_A; f = f.$$

   - Associative law. If $f : A \longrightarrow B, g : B \longrightarrow C$ and $h : C \longrightarrow D$ then

   $$(f; g); h = f; (g; h) : A \longrightarrow D.$$

The **charity** language is set in the distributive formal category of datatypes (Cockett & Fukushima, 1992).

2.2 Initial Datatypes

Initial datatypes, also known as *inductive* or *left* datatypes, are defined

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

where A is taken to be a power of other datatypes.

$L(A)$ is the new datatype and contains the maps $c_i : E_i(A, L(A)) \longrightarrow L(A)$ whose type is obtained by setting $S = L(A)$ in the definition. These maps are called constructors of the type construction L.

2.3 Final Datatypes

The dual notion to initial datatypes is *final* datatypes, also known as *coinductive* or *right* datatypes. These are defined

$$\begin{aligned} \text{data } C \to R(A) \quad = & \\ & d_1 : S \to F_1(A, S) \\ & \quad \vdots \\ | \quad & d_n : S \to F_n(A, S) \end{aligned}$$

where A is taken to be a power of other datatypes.

The new datatype, $R(A)$, contains the maps $d_i : R(A) \to F_i(A, R(A))$ whose type is obtained by setting $S = R(A)$ in the definition. These maps are called destructors and R is called the type constructor of the datatype.

2.4 Basic Charity Functions

All functions in **charity** work on either inductive or coinductive datatypes as defined above. When a datatype is defined in **charity** some basic functions come with it for free. The most fundamental functions of these are the fold, for inductive datatypes, and the unfold, for coinductive datatypes. All other functions may be defined from these two. However, for convenience, some other functions are also given as basic. These are the case function for initial datatypes, the record for final datatypes, and the map for both initial and final datatypes. All programmer defined functions are built from these basic functions.

2.5 Structure of Charity

At this point, before moving on to examine these basic functions, it is worth looking at how **charity**, the language is structured. As we have just seen, the basis for the language is a category of datatypes. Programs are expressed in terms of categorical combinators. A combinator is a function which produces new functions from other functions. A program in **charity** is simply a string of combinators arranged such that their domains and codomains match up. Evaluation is done through rewrite rules which specify how certain strings of combinators can be reduced. When all possible rewrites are completed the program has been evaluated. Unfortunately, it is very difficult to write programs at the combinator level. For instance

```
pair{zero{}; succ{}, zero{}; succ{}}; foldnat{p1{}, p0{}; succ{}}
```

is how to add one plus one (Cockett, 1993b). So a *term logic*[1] has been developed to more naturally express functions. The term logic is directly translatable into categorical combinators (Cockett & Fukushima, 1992). In the term logic adding one and one is done

```
{| zero : () => succ(zero)
 | succ : x  => succ(x)
 |}(succ(zero)).
```

Unless the reader is already familiar with **charity** programming this is not likely to be any more readable than the combinator level example! This example is explained in section 2.7.3. In the next section we will describe programming in **charity** at the term logic level.

2.6 Programming in Charity

Like many functional languages **charity** is an interpreted language. The examples in the next sections will be displayed as if they were being programmed in the interpreter. After starting **charity** the prompt is displayed:

```
>
```

The first thing to do is to build some datatypes.

2.7 Inductive Datatypes and Their Functions

2.7.1 Building Inductive Datatypes

One useful datatype would be

[1]Actually, the combinators are also a term logic for the mathematical notation of categories. We will not use term logic in this manner however.

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

This says that the type `bool` has two constructors, `false` and `true`, both of which are maps from the initial object, `1`, to `bool`.

Now, we will make a datatype to represent the natural numbers.

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

The `nats` datatype is a little more interesting since the constructor `succ` is a map from `nats` to itself. The `nats` represent natural numbers in a straightforward way:

0	is	`zero`
1	is	`succ(zero)`
2	is	`succ(succ(zero))`
3	is	`succ(succ(succ(zero)))`
⋮		⋮

2.7.2 The Case

The *case* is used to make choices between constructors of a datatype. To check if a natural number is zero:

```
> { zero    => true
  | succ(_) => false
  } (succ(zero)).
false : bool
11 machine operations,  0.0 cpu sec
```

The case expression is delimited by `{` and `}`. The term `succ(zero)` is applied to it and the interpreter returns `false` (of type `bool`) as the answer. The list of constructors in the case expression must be complete. That is, any possible input to the case expression must be anticipated and provided for. Note the use of the *don't care* symbol (_). The constructor `succ` takes an argument, so *something* had to be put in its argument base. But since we were not going to refer to that argument again we used the don't care symbol. It can be thought of as an anonymous variable.

Although the above case works fine, it is not very useful since we have to reenter it every time we want to apply it to a different number.

```
> def is_zero(n) = { zero    => true
                   | succ(_) => false
                   }(n).
is_zero  : nat -> bool
```

defines a function that will perform the zero check. Now, we need only enter

```
> is_zero(succ(zero)).
false : bool
11 machine operations,  0.0 cpu sec
```

or

```
> is_zero(zero).
true : bool
10 machine operations,  0.0 cpu sec
```

et cetera.

2.7.3 The Fold

The natural numbers can be added together using a fold. A function that adds two natural numbers is

```
> def add(m, n) = {| zero : () => m
                   | succ : x  => succ(x)
                   |} (n).
add  : nat * nat -> nat
```

The fold expression is delimited by `{|` and `|}` or *barbed wire.*

The fold can be thought of as replacing the constructors in the data being folded over, in this case it is `n`, with the term that appears on the right hand side of the constructor in the fold. This fold says:

Replace the `zero` in `n` with `m`.

Replace every occurrence of `succ` in `n` with `succ`.

Function `add` gives results like

```
> add(succ(succ(zero)), succ(zero)).
succ(succ(succ(zero))) : nat
27 machine operations,  0.0 cpu sec
```

2.7.4 Lists and Polymorphic Types

The most important datatype in functional programming is the list.

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

As can be seen from the datatype declaration the `list` datatype allows the programmer to group data of the same type together (in a list). The list endpoint is indicated by a `nil`. Thus, a list of `nats` looks like:

```
> cons(succ(zero), cons(zero, cons(succ(succ(zero)), nil))).
[succ(zero),zero,succ(succ(zero))] : list(nat)
24 machine operations,  0.010000 cpu sec
```

Note that **charity** represented the list as `[succ(zero),zero,succ(succ(zero))]`. This convenient shorthand is available to the programmer also.

The list type declaration illustrates another important aspect of functional languages in general and **charity** in particular. This is the *polymorphic* type. `list` is a parameterized datatype where `A` is the parameter. The type of `A` is *not* given in the `list` type declaration; it is determined when a program is compiled. Thus, in addition to having a list of `nats`, we can also have lists of `bools`

```
> cons(true, cons(false, nil)).
[true,false] : list(bool)
15 machine operations,  0.0 cpu sec
```

This means the programmer can write generic functions for lists. For instance, the following function determines the length of a list:

```
> def length(lst) = {| nil  : ()      => zero
                     | cons : (_, x) => succ(x)
                     |} (lst).
length  : list('a) -> nat
```

The fold says

> Replace the `nil` in `lst` with `zero`.
>
> Replace every occurrence of `cons` in `lst` with `succ`.

Note the type of the function. The list is of some indeterminate type. This means we can call it with a list of type `bools`, of type `nats`, or of any other type.

```
> length([succ(zero),zero,succ(succ(zero))]).
succ(succ(succ(zero))) : nat
70 machine operations,  0.010000 cpu sec
> length([true,false]).
succ(succ(zero)) : nat
48 machine operations,  0.010000 cpu sec
```

2.7.5 The Map

The final basic functional building block for inductive datatypes is the *map*. The map is a very useful function. It operates over a specified datatype, replacing its parameters with new one. For instance, to add one to a list of `nats`, replace the original values with values one larger.

```
> def inc_list(lst) = list{x => succ(x)}(lst).
inc_list  : list(nat) -> list(nat)
```

Notice how the compiler realized the list was of type `list(nat)` from the presence of the `succ(x)` term. Function `inc_list` is used like this:

```
> inc_list([succ(zero),zero,succ(succ(zero))]).
[succ(succ(zero)),succ(zero),succ(succ(succ(zero)))] : list(nat)
51 machine operations,  0.010000 cpu sec
```

A list of `nats` can be changed to a list of `bools` such that, if the member of `list(nat)` is `zero` then it becomes `true` otherwise it becomes `false`:

```
> def make_bool_list(natlst) = list{x => is_zero(x)}(natlst).
make_bool_list  : list(nat) -> list(bool)
```

Then

```
> make_bool_list([succ(zero),zero,succ(succ(zero))]).
[false,true,false] : list(bool)
69 machine operations,  0.010000 cpu sec
```

This function also illustrates another useful property of functional languages; that of combining functions to make new functions.

2.7.6 Macros in Charity

We now use lists to illustrate the use of macros in **charity** . Suppose we have another function

```
> def add_one(n) = {x => succ(x)}(n).
add_one  : nat -> nat
```

and another map function

```
>  def make_new_list{f}(lst) =list{x => f(x)}(lst).
make_new_list {'a -> 'c} : list('a) -> list('c)
```

The `f` is standing in for a function which is specified when the function is called. For example,

```
> make_new_list{x => add_one(x)}([succ(zero),zero,succ(succ(zero))]).
[succ(succ(zero)),succ(zero),succ(succ(succ(zero)))] : list(nat)
85 machine operations,  0.020000 cpu sec
```

```
> make_new_list{x => is_zero(x)}([succ(zero),zero,succ(succ(zero))]).
[false,true,false] : list(bool)
91 machine operations,  0.020000 cpu sec
```

In general, functions in **charity** cannot call themselves and macros are also subject to this restriction. This is done to prevent non-termination. Macros do not give **charity** higher order functions. A language with higher order functions allows functions that produce other functions as output and also allows functions to be passed freely to other functions as if they were data. **Charity** allows neither.

2.7.7 A Final Inductive Datatype Example

Our final example of an inductive datatype function determines if two nats are equal (Cockett, 1993a).

```
> def equal_nats(m, n) = { (flag, num) => {zero    => flag
                                          |succ(_) => false
                                          }(num)
                         } ( {| zero : ()     => (true, m)
                              | succ : (_, x) => { zero     => (false, zero)
                                                 | succ(x') => (true, x')
                                                 } (x)
                              |} (n) ).
equal_nats  : nat * nat -> bool
```

The fold expression says:

> Replace the `zero` in `n` with `(true, m)`.
>
> Replace every `succ` in `n` with `{ (_, x) => {zero     => (false, zero)`
> `|succ(x') => (true, x')`
> `} (x) }`.

Thus, every `succ` decrements `m` by one until, or if, it reaches `zero`. If it has reached `zero` by the last `succ` in `n` then `(false, zero)` is returned. Otherwise, `(true, x')` is returned, where `x'` is what is left of `m`.

So, only if `m` was equal to `n` would the pair `(true, zero)` be returned. The case will only return `true` in this event, in all others it will return `false`. We will return to this example when we discuss pattern matching.

2.8 Coinductive Datatypes and Their Functions

2.8.1 Building Coinductive Datatypes

The quintessential right datatype is declared

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

2.8.2 The Unfold

Unlike constructors we cannot immediately begin to use destructors to make data. This is because destructors are mapping *from* the new datatype to another type. Thus we must make the data for the datatype first and then use the destructors to find out what it is. For this we will use the *unfold*. The unfold expression is delimited by `(|` and `|)`; called *lenses*. For instance, an infinite list of `nats` beginning at `zero` is

```
> def infnats = (| x => head : x
                 |      tail : succ(x)
                 |) (zero).
infnats  : 'a -> stream(nat)
```

In this unfold the destructor `head` is a map from `stream` to `nat`. The destructor `head` will return whatever term, or state, is being applied to the unfold. The destructor `tail` is a map from `stream` to `stream`. Thus, it determines a new state (in this case incrementing `x`) to apply to the unfold and returns the application.

To find out what the elements of `infnats` are

```
> infnats.
... : stream(nat)
8 machine operations,  0.0 cpu sec
```

```
Entering display-right-data mode...

(head:zero,tail:...)
Press <Return> to display more or ‘q’ to quit:

(head:zero,tail:(head:succ(zero),tail:...))
Press <Return> to display more or ‘q’ to quit:

(head:zero,tail:(head:succ(zero),tail:(head:succ(succ(zero)),tail:...)))
Press <Return> to display more or ‘q’ to quit:

(head:zero,tail:(head:succ(zero),tail:(head:succ(succ(zero)),
                           tail:(head:succ(succ(succ(zero))),tail:...))))
Press <Return> to display more or ‘q’ to quit: q

>
```

or, alternatively, the destructors can be applied directly to the unfold

```
> head(infnats).
zero : nat
11 machine operations,  0.010000 cpu sec
> head(tail(infnats)).
succ(zero) : nat
20 machine operations,  0.010000 cpu sec
> head(tail(tail(infnats))).
succ(succ(zero)) : nat
29 machine operations,  0.0 cpu sec
> head(tail(tail(tail(infnats)))).
succ(succ(succ(zero))) : nat
38 machine operations,  0.010000 cpu sec
```

As can be seen, **charity** does not fully evaluate the infinite stream immediately! It only evaluates what is necessary to be able to satisfy the current demand. This is an example of *lazy evaluation*.

2.8.3 The Record

A *record* is used to add elements to the beginning of an infinite data type. For instance, to prepend two `zeros` to `infnats`

```
(head : zero, tail : (head : zero, tail : infnats))
```

2.8.4 The Map (Again)

Maps can be used on infinite data types also. One way to create an infinite matrix of natural numbers is

```
stream{ x => make_infnats(x) } (make_infnats(zero)).
```

This maps every element of the stream `infnats` to an infinite stream of `nats`. The result can be pictured

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & \dots \\ 1 & 2 & 3 & 4 & \dots \\ 2 & 3 & 4 & 5 & \dots \\ 3 & 4 & 5 & 6 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

2.8.5 A Final Coinductive Datatype Example

As a final example of infinite datatypes consider the conatural numbers (Cockett, 1993a)

```
data C -> conats = denat: C -> 1 + C.
```

The natural numbers may be embedded in the conatural numbers by

```
def make_conats(n) = (| x => denat: {zero    => b0()
                                     |succ(m) => b1(m)
                                     }(x)
                     |)(n).
```

The above also illustrates the use of coproducts. The `+` is infix notation for the datatype

```
> data +(A,B) -> C =
        b0 : A -> C
      | b1 : B -> C.
```

Thus, the `conats` datatype definition defined the `denat` as a map from some datatype `C` to either `1` or C. In `make_conat` we see that `b0()` specifies the `1` type while `b1(m)` specifies the state to return to the unfold.

Conats may be added.

```
def coadd(m,n) =
        (| (x,y) => denat : {b0() => {b0()  => b0()
                                     |b1(n) => b1((denat:b0()), n)
                                     } (denat(y))
                            |b1(m) => b1(m,y)
                            } (denat(x))
         |) (m,n).
```

We will return to this example when doing pattern matching.

Chapter 3

Pattern Matching

Pattern matching is a way of making implicit case statements. This makes code clearer and less difficult to write. It does not add any computational power to a language; indeed, it may cause some loss in speed as the compiler must make the implicit case statements explicit. Yet, every major functional language such as SML and Miranda has pattern matching. In this chapter are some motivational examples of what pattern matching can do. In chapter 4 , patterns will be explained generally in terms of **charity** and Miranda. Then, in chapter 6, patterns will be precisely defined in terms of the **charity** language.

3.1 Pattern Matching Examples

The `and` relation could be expressed

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

With pattern matching we can write the case expression as if we were casing on the `(x, y)` pair.

```
> def and'(x,y) = { (true, true) => true
                  | (true, false) => false
                  | (false, _    ) => false
                  }(x,y).
and'  : bool * bool -> bool
```

or, even more succinctly,

```
> def and''(x,y) = { (true, true) => true
                   |      _        => false
                   }(x,y).
and''  : bool * bool -> bool
```

Even though the syntactical look of the `and'` and `and''` functions has changed from the non-pattern matched form, it is still evaluated in the same manner.

Pattern matching can be used in fold expressions also. Recall the `equal_nats` function given in section 2.7.7. This is the same function written with pattern matching:

```
> def equal_nats(m, n) = { (flag, zero) => flag
                         |      _       => false
                         } ( {| zero : ()              => (true, m)
                              | succ : ( _ , zero)     => (false, zero)
                                     | ( _ , succ(x)) => (true, x)
                              |} (n))
equal_nats : nat * nat -> nat
```

The logic is much clearer here!

In section 2.8.5 a function to add `conats` was given. The same function written using pattern matching illustrates the use of pattern matching in unfolds.

```
> def coadd(m,n) = (| (b0(),  b0())  => denat : b0()
                    | (b0(),  b1(n)) => denat : b1((denat : b0()), n)
                    | (b1(m), n)     => denat : b1(m, (denat : n))
                    |) (denat(m), denat(n)).
coadd  : conats * conats -> conats
```

Again, this is much easier to read and understand.

3.2 An Extended Pattern Matching Example

Of course, the more decisions one must make, the greater the number of nested case statements in a language without pattern matching. This situation can be encountered if one is attempting to keep track of a state. Following is a slightly contrived example to illustrate this.

We are going to build a program that will parse and, at the same time, evaluate a list of tokens consisting of booleans and boolean operators. The datatype of the tokens is

```
data tokens -> C =
         T   : 1 -> C
        |F   : 1 -> C
        |AND : 1 -> C
        |OR  : 1 -> C
        |NEG : 1 -> C.
```

We will use the success or fail datatype

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

to return the answer. If there are no parse errors then `ss(bool)` will be returned, otherwise `ff` will be returned.

The grammar we will attempt to parse is very simple

$$\boxed{\begin{array}{rcl} \mathit{expr} & \to & \mathit{boolval} \mid \mathit{boolval\ partexp} \\ \mathit{partexp} & \to & \mathit{boolop\ boolval} \mid \mathit{boolop\ boolval\ partexp} \\ \mathit{boolval} & \to & \mathrm{T} \mid \mathrm{F} \mid \mathrm{NEG\ T} \mid \mathrm{NEG\ F} \\ \mathit{boolop} & \to & \mathrm{AND} \mid \mathrm{OR} \end{array}}.$$

The program works by folding over the length of the token list[1]. The initial state is a pair consisting of the boolean value of the first *boolval* in the token list, and the rest of the token list. Thereafter, each subsequent fold operation removes a *boolop* and a *boolval* from

[1]This is approximately twice as long as necessary but it will not affect the result of the program.

the token list, evaluates the result so far, and returns the current evaluation and the rest of the token list.

This is the pattern matched version of the program

```
def dostart(expr) = {cons(T, expr')            => (ss(true),  expr')
                    |cons(F, expr')            => (ss(false), expr')
                    |cons(NEG, cons(T, expr')) => (ss(false), expr')
                    |cons(NEG, cons(F, expr')) => (ss(true),  expr')
                    |         _                => (ff, nil)
                    }(expr).


def parse(r, expr) =
  p0({| zero:  ()       => dostart(expr)
      | succ:  (r, nil) => (r, nil)
              |(cons(AND, cons(T, expr')), ss(true))            => (ss(true) , expr')
              |(cons(AND, cons(T, expr')), ss(false))           => (ss(false), expr')
              |(cons(AND, cons(F, expr')),     _    )           => (ss(false), expr')
              |(cons(AND, cons(NEG, cons(T, expr'))), _ )       => (ss(false), expr')
              |(cons(AND, cons(NEG, cons(F, expr'))), ss(true))  => (ss(true) , expr')
              |(cons(AND, cons(NEG, cons(F, expr'))), ss(false)) => (ss(false), expr')
              |(cons(OR, cons(T, expr')), _ )                   => (ss(true) , expr')
              |(cons(OR, cons(F, expr')), ss(true))             => (ss(true) , expr')
              |(cons(OR, cons(F, expr')), ss(false))            => (ss(false), expr')
              |(cons(OR, cons(NEG, cons(T, expr'))), ss(true))   => (ss(true) , expr')
              |(cons(OR, cons(NEG, cons(F, expr'))), ss(false))  => (ss(false), expr')
              |(cons(OR, cons(NEG, cons(F, expr'))), _)         => (ss(true) , expr')
              |    _                                            => (ff       , nil)
    }(length(expr)).
```

Here is some sample input and output

```
>  parse( [ T, AND, F]).

ss(false) : ss_ff(bool)

221 machine operations,  0.090000 cpu sec

> parse( [ NEG, F, OR, NEG, T, AND, T]).

ss(true) : ss_ff(bool)

525 machine operations,  0.120000 cpu sec

> parse( [ OR, T ]).

ff : ss_ff(bool)

114 machine operations,  0.080000 cpu sec
```

The `doeval` function of the same program written without pattern matching is given in figure 3.1. The logic is not apparent at all. It should be obvious this version of the program would be much harder to write, maintain, and understand. The full text of the non-pattern matched program is given in Appendix A.

```
def doeval(r, expr) =
  {nil()          => (r,nil)
  |cons(tk1,expr') =>
     {AND() => {nil()           => (ff, nil)
               |cons(tk2,expr'') =>
                  {T()   => {ff()  => (ff, nil)
                            |ss(b) => {true()  => (ss(true) ,expr'')
                                      |false() => (ss(false),expr'')
                                      }(b)
                            }(r)
                  |F()   => (ss(false),expr'')
                  |NEG() => {nil()            => (ff, nil)
                            |cons(tk3,expr''') =>
                               {T() => (ss(false),expr''')
                               |F() => {ff()  => (ff, nil)
                                       |ss(b) =>
                                           {true() => (ss(true),expr''')
                                          |false() => (ss(false),expr''')
                                          }(b)
                                       }(r)
                               |_   => (ff, nil)
                               }(tk3)
                            }(expr'')
                  | _    => (ff, nil)
                  }(tk2)
                }(expr')
     |OR()  => {nil()           => (ff, nil)
               |cons(tk2,expr'') =>
                  {T()   => (ss(true),expr'')
                  |F()   => {ff()  => (ff, nil)
                            |ss(b) => {true()  => (ss(true) ,expr'')
                                      |false() => (ss(false),expr'')
                                      }(b)
                            }(r)
                  |NEG() => {nil()            => (ff, nil)
                            |cons(tk3,expr''') =>
                               {T() => {ff()  => (ff, nil)
                                       |ss(b) =>
                                           {true() => (ss(true),expr''')
                                          |false() => (ss(false),expr''')
                                          }(b)
                                       }(r)
                               |F() => (ss(true),expr''')
                               |_   => (ff, nil)
                               }(tk3)
                            }(expr'')
                  | _    => (ff, nil)
                  }(tk2)
                }(expr')
     | _    =>        (ff, nil)
     }(tk1)
  }(expr).
```

Figure 3.1: Function **doeval** with no pattern matching

Chapter 4

Pattern Matching In Functional Languages

Pattern matching in **charity** is precisely defined in chapter 6. Other functional languages may have different types of pattern matching. This chapter looks at several common types of patterns and discusses them in relation to **charity** and Miranda.

4.1 Types of Pattern Matching

Some common types of patterns are (Peyton Jones, 1987):

- overlapping patterns
- constant patterns
- nested patterns
- multiple arguments
- non-exhaustive sets of equations
- conditional equations
- repeated variables

Each type of pattern matching will be explained in the following subsections.

4.1.1 Overlapping Patterns

Overlapping patterns are a series of patterns that can be matched with a non-disjoint set of patterns. In Miranda and most other functional languages evaluation depends upon the syntactical ordering of the patterns; in other words the patterns are checked from top to bottom and the first one that matches is used. An example of overlapping patterns is:

```
def foo(x,n) =  {zero => x
                | _    => succ(x)
                }(n).
```

If the ordering of the cases was reversed in this example the `zero` case would never be evaluated because it would be *completely overlapped* by the "_" case. Then the `zero` pattern would be *redundant*. Generally the presence of a redundant pattern indicates an oversight on the part of the programmer or a logic flaw in the design of the program. A redundant pattern is not an error but a warning message should be produced by the compiler. **Charity** allows overlapping patterns (see section 6.1.3). It will check for redundant patterns by checking to see that all phrases are used when the pattern matched term logic is translated.

4.1.2 Constant Patterns

Peyton-Jones used constant in the sense of a literal constant such as a character, character string or number. In this sense, **charity** does not have constants since everything is developed from datatypes. However, **charity** does have constants in the sense that there exists patterns like `zero` or `succ(zero)` that mean 0 and 1. Indeed, the integers $0, 1, 2, \ldots$ can be defined as constructors.

Thus, patterns such as those contained in this case statement,

```
def foo(x) = {1 => 2
             |2 => 3
             |x => x }(x).
```

are acceptable in **charity** and Miranda.

4.1.3 Nested Patterns

Patterns are *nested* when one (or more) patterns are contained within another pattern. As an example consider:

```
cons(zero, cons(succ(zero), nil))
```

The constructor patterns `zero` and `cons(succ(zero), nil)` are nested inside the first `cons`. The patterns `succ(zero)` and `nil` are nested inside the second `cons`.

Both Miranda and **charity** support this concept. Nested patterns can easily lead the programmer to write non-exhaustive patterns (see section 4.1.5) which are not allowed in **charity**.

4.1.4 Multiple Arguments

If a function takes more than one argument then each argument can be a pattern. This nested case statement demonstrates this:

```
def foo =   (succ(zero), cons(zero,nil)) => true
           |(succ(zero),  _)             => false
           |            _                => true.
```

Multiple argument patterns are allowed in both **charity** and Miranda.

4.1.5 Non-exhaustive Patterns

There is the possibility an argument can pass type checking[1] yet fail to match any of a function's patterns. This is the situation for the case expression given below:

```
{ (zero,    nil)           => cons(false, nil)
| (succ(x), cons(x, L))    => L
}.
```

There are two cases missing; this next case expression has them filled in:

[1]Checking for a complete set of patterns is not considered part of the typechecking. Pattern completeness is checked during the translation from extended term logic to core term logic (see section 8.1).

```
{ (zero,    nil)          => [false]
| (succ(x), cons(x, L))   => L
| (zero,    cons(x, L))   => [false, true]
| (succ(_), nil)          => [true, false]
}.
```

This is an *exhaustive* or complete set of patterns.

Charity requires that all patterns be of the *exhaustive* variety (see chapter 6). Miranda, though, has no problem with non-exhaustive patterns until you pass the function an argument that does not match; then it will give a runtime error.

4.1.6 Conditional Equations

The *conditional equation* type of pattern matching arises when the programming language allows the programmer to *guard* the application of a pattern with a conditional equation. This can lead to a non-exhaustive set of patterns since the guard may screen out a pattern that would otherwise match. Miranda allows both guards and non-exhaustive pattern matching using guards. The following is an example of a conditional equation written in Miranda syntax:

```
max x y z
  = x,  if x > y & x > z
  = y,  if y > z
  = z
```

Adding conditional equations to **charity** is not within the scope of this project for two reasons. Firstly, adding guards to patterns will require additional syntactic extensions beyond what is required for pattern matching. Secondly, since **charity** has been designed to not succumb to runtime errors some method of ensuring the guards cannot cause non-exhaustive patterns will be required.

4.1.7 Repeated Variables

The repeated use of a variable within a pattern implies that each use of the *repeated variable* is equivalent to each other use. As an example consider:

```
def foo(List) =
            {|nil : () => nil
             |cons: ((x,x),L) => cons(x,L)
             |     | ((_,_),L) => L
             |}(List).
```

The pattern `((x,x),L)` contains the variable `x` repeated once. The problem here is that **charity** needs to be able to test for equality on arbitrary datatypes. In general, this is not possible since the datatypes could be infinite; therefore, it is not allowed in **charity**. Miranda allows this type of pattern matching.

Chapter 5

Project Outline

Chapters 6 through 8 detail the work done so far to implement pattern matching in **charity**. This chapter is intended to provide some context for these later chapters by giving an outline of what is required to fully implement pattern matching in **charity**.

5.1 Theoretical Aspects

Before implementation could begin, some theoretical details had to be worked out. A precise definition for patterns in the term logic had to be formulated. This also involved defining what was meant by a complete set of patterns, and what it meant to say that one set of patterns was more general than another set.

Since patterns were replacing variable bases in the term logic, any terms that used variable bases in the term logic had to be redefined using patterns. This included redefining what is meant by a program. From the new term definitions the programming syntax of the pattern matching version of **charity** was developed. At this point, it was possible to implement the syntax and this was done.

As is common with most functional languages, the **charity** term logic must be translated to combinators before it can be evaluated. This translation already exists for the current term logic, but not for the newly developed pattern matching term logic. The most straight forward approach to evaluating the pattern matching term logic is to translate it to the current term logic, and then translate the current term logic to combinators as before. Therefore, a translation algorithm to do this was required. This was refined and developed from existing algorithms (see section 8.1).

While not strictly required for the implementation of pattern matching, it was decided to implement type checking in the term logic as part of this project for reasons discussed in section 7. Again, an algorithm was required to do this. Algorithms exist in the literature but they had to be adapted to the specific needs of this project.

Finally, the generated core term logic code can be optimized using decision tree analysis techniques. Although this will be done in the future, it was not addressed in the current project.

5.2 Implementation Details

Pattern matching will be added to v3.11 of **charity**. This version was written by Tom Fukushima in the language SML. Currently, there are two other versions of **charity** around. One version, written in C, was done as a CPSC 502 project by Marc Schroeder and Barry Yee (Schroeder, 1993; Yee, 1993). The other version has monad capability and is another SML language program written by Fukushima.

It was decided to implement pattern matching on v3.11 because it was a stable version that had been thoroughly tested. In addition, it was written in a functional language so development time would be quicker. However, it is planned to include pattern matching in both the C language version and the monad version of **charity** in the future.

Implementing pattern matching in the **charity** language involves four steps. First, the parser must be changed to accept the pattern matching syntax. After a successful parse, the term logic needs to be typechecked. Third, the pattern matched terms must be translated into core term logic terms. Finally, the resulting core term logic terms should be optimized.

SML-Yacc (Tarditi & Appel, 1991) was used to build the parser for v3.11. The yacc program was also used for this project with modifications made to the original grammar specification as required. The term logic portion of the original yacc specification and the extended specification are in Appendices D and E respectively.

Typechecking will be performed immediately after the parsing of a term or function is complete. This is done in two different places in the compiler depending on whether a function or a term was found.

Translation and optimization take place immediately after typechecking is complete.

From there the translation to categorical combinators takes place as with the core term logic.

Chapter 6

Patterns In Charity

We will refer to the current **charity** term logic as defined in *About Charity* (Cockett & Fukushima, 1992) as the *core term logic*. The pattern matching term logic, defined in this chapter, will be referred to as the *extended term logic*.

The extended term logic does not preclude programs or terms expressed in core term logic. Variable bases (see appendix B) may be described as patterns and any term in the core term logic may also be expressed in the extended term logic. This is important since it means any previously developed code will still work in the pattern matching **charity** language.

The syntax of the pattern matching charity language is derived directly from the mathematical expression of its patterns, terms and programs (as was the case with the core **charity** language). A description of the changes required to support pattern matching syntax is given in section 6.3. A complete description of the grammar for the term logic portion of extended **charity** is given in appendix E.

6.1 Patterns

The variable bases in core **charity** term logic are replaced by patterns in the pattern matching term logic. This removes the need for the programmer to explicitly declare case statements to direct the result of a program or term; this logic is now implicit in the pattern (as we have seen in chapter 3).

Patterns must be complete. By this it is meant that any possible, well formed input must match at least one pattern. This preserves **charity**'s guarantee of (proper) termination. It also makes it possible to translate the extended term logic into the core term logic.

6.1.1 Definition of Patterns

A pattern is defined in **charity** by:

- $()$ is a pattern of type 1,
- if v is a variable then v is a pattern of type type(v),
- if p_0 and p_1 are patterns with *no* variables in common then (p_0, p_1) is a pattern where
$$\text{type}((p_0, p_1)) = \text{type}(p_0) \times \text{type}(p_1),$$
- if c_i is the ith constructor of data type $L(A)$, and p_i is a pattern where type$(p_i) = E_i(A, L(A))$, then $c_i(p_i)$ is a pattern of type $L(A)$,
- if $d_1, \ldots, d_n$ are destructors of the coinductive datatype R(A) where
$$S \mapsto R(A) = d_i : S \mapsto F_i(A, R(A))$$
,
and $p_1, \ldots, p_n$ are patterns with *no* variables in common where
$$\text{type}(p_1) = F_1(A, R(A)), \ldots, \text{type}(p_n) = F_n(A, R(A))$$
then $(d_1 : p_1, \ldots, d_n : p_n)$ is a pattern of type $R(A)$.

6.1.2 More General Patterns

Let p and p' be patterns of the same type and let v be a variable pattern. Let p' is more general than p be denoted by $p' \gg p$. Then $p' \gg p$ can be defined inductively as:

- $v \gg ()$,
- $v \gg (p_1, p_2)$,
- $v \gg c_i(p_i)$,
- $v \gg (d_1 : p_1, \ldots, d_n : p_n)$;

if $q' \gg q$ then

- $(q', p_2) \gg (q, p_2)$,
- $(p_1, q') \gg (p_1, q)$,

- $c_i(q') \gg c_i(q)$,
- $(d_1 : p_1, \ldots, d_i : q', \ldots, d_n : p_n) \gg (d_1 : p_1, \ldots, d_i : q, \ldots, d_n : p_n)$, where $1 \le i \le n$.

6.1.3 Complete Sets of Patterns

A complete set of patterns, P, of type X will be denoted $P \triangleleft X$. This relation can be defined inductively as follows:

- $() \triangleleft 1$,
- $\{v\} \triangleleft X$, where v is a variable of type X,
- if $P \triangleleft X$ and $P' \triangleleft Y$ then $\{(p, p') \mid p \in P, p' \in P'\} \triangleleft X \times Y$,
- if $c_i{:}E_i(A, L(A))$, $1 \le i \le n$, are the constructors of $L(A)$ then, if, for each $P_i \triangleleft E_i(A, L(A))$, $\bigcup_{i=1\ldots n} \{c_i(p) \mid p \in P_i\} \triangleleft L(A)$,
- if $P_i \triangleleft F_i(A, R(A))$ then $\{(d_1{:}p_1, \ldots, d_n{:}p_n) \mid p_1 \in P_1, \ldots, p_n \in P_n\} \triangleleft R(A)$.

A set of patterns may be more general than another set of patterns. Let P' be the set of patterns formed from P by substituting at least one $p \in P$ with p', where $p' \gg p$. Then $P' \gg P$.

It is possible to add patterns to a complete set of patterns and still have it remain complete. It is also possible to replace patterns in a complete set with other patterns and still have the set remain complete, as long as the new set of patterns is more general than the old set. The rules for doing so are given below:

- if $\text{type}(p) = X$ and $P \triangleleft X$ then $\{p\} \cup P \triangleleft X$
- if $P' \gg P$, and $P \triangleleft X$ then $P' \triangleleft X$

6.2 Pattern Matching Terms

The following term definitions come from a technical report written by Cockett and Fukushima (Cockett & Fukushima, 1992), with the exception of the case term, fold term, map term, and unfold term. These are redefined here for pattern matching. The core term logic definitions of these terms are given in Appendix B.

6.2.1 Definition of Terms

A **term** is defined by:

- $()$ is a term of type $\mathrm{type}(()) = 1$;
- if t is a term where $\mathrm{type}(t) = X \times Y$ then $\mathrm{p}_0(t)$ and $\mathrm{p}_1(t)$ are terms where $\mathrm{type}(\mathrm{p}_0(t)) = X$ and $\mathrm{type}(\mathrm{p}_1(t)) = Y$;
- if t_0 and t_1 are terms then (t_0, t_1) is a term where $\mathrm{type}((t_0, t_1)) = \mathrm{type}(t_0) \times \mathrm{type}(t_1)$;
- if t is a term then $\mathrm{b}_0(t)$ and $\mathrm{b}_1(t)$ are terms where $\mathrm{type}(\mathrm{b}_0(t)) = \mathrm{type}(t) + X$ and $\mathrm{type}(\mathrm{b}_1(t)) = X + \mathrm{type}(t)$ (where X is an indeterminate type);
- if $t_1, \ldots, t_n$ are terms and $\forall i$ such that $1 \leq i \leq n$ it is the case that $\mathrm{type}(t_i) = E_i(A, L(A))$ then $c_1(t_1), \ldots, c_n(t_n)$ are terms where $\mathrm{type}(c_i(t_i)) = L(A)$;
- if t is a term where $\mathrm{type}(t) = P$, and $\{p_1, \ldots, p_m\} \lhd P$, and $t_1, \ldots, t_m$ are all terms of type B then

  $$\left\{ \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array} \right\}(t)$$

  is a term (the case term) of type B; any variables in p_i are bound in t_i;
- if t is a term where $\mathrm{type}(t) = L(A)$, and c_i is a constructor of data type $L(A)$, and $\{p_{i1}, \ldots, p_{im_i}\} \lhd E_i(A, X)$, and $\forall ij \cdot t_{ij}$ is a term of type X then

  $$\left\{\left| \begin{array}{ll} c_1 : & \left| \begin{array}{ccc} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1} \end{array} \right. \\ \vdots & \\ c_n : & \left| \begin{array}{ccc} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n} \end{array} \right. \end{array} \right|\right\}(t)$$

  is a term (the fold) of type X; any variables in p_{ij} are bound in t_{ij};

- if t is a term where type$(t) = L(A_1, \ldots, A_n)$, and $\{p_{i1}, \ldots, p_{im_i}\} \lhd A_i$, and $t_{i1}, \ldots, t_{im_i}$ are terms of type B_i then

$$L\left\{\begin{array}{l}\left|\begin{array}{lll} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1}\end{array}\right. \\ \vdots \\ \left|\begin{array}{lll} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n}\end{array}\right.\end{array}\right\}(t)$$

is a term (the map) of type $L(B_1, \ldots, B_n)$; any variables in p_{ij} are bound in t_{ij};

- if t is a term where type$(t) = S$, and $\{p_1, \ldots, p_m\} \lhd S$, and $t_{i1}, \ldots, t_{in}$ are terms of type $F_i(A, S)$ then

$$\left(\!\left|\begin{array}{lll} p_1 & \mapsto & \left|\begin{array}{lll} d_1 & : & t_{11} \\ & \vdots & \\ d_n & : & t_{1n}\end{array}\right. \\ & \vdots & \\ p_m & \mapsto & \left|\begin{array}{lll} d_1 & : & t_{m1} \\ & \vdots & \\ d_n & : & t_{mn}\end{array}\right.\end{array}\right|\!\right)(t)$$

is a term (the unfold) of type $R(A)$; any variables in p_i are bound in t_{ij};

- if $t_1, \ldots, t_n$ are terms where type$(t_i) = F_i(A, R(A))$ then

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

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

6.2.2 Pattern Matching Programs

A program, or function, in **charity** is not a term but a closed abstraction. If $\{p_1, \ldots, p_m\} \lhd P$, and $t_1, \ldots, t_m$ are all terms of type B then

$$\left\{ \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array} \right\},$$

is an abstraction of type $P \longrightarrow B$, and, when p_i contains all the free variables in t_i, it is closed.

6.3 Programming Syntax

Obviously, the forms for the pattern matching terms and programs given above cannot be used directly to build **charity** language programs on a computer. This section gives a programming syntax for each term that can be parsed by the **charity** compiler. The syntax for the terms that were altered by the addition of pattern matching are given below. For a complete description of the **charity** programming system the reader should consult the *Charity User's Manual* (Fukushima, 1991).

- A pattern is:

pattern $\rightarrow$

()

| (p)

| (*pattern*, *pattern*)

| *variable*

| c(*pattern*)

| (d_1 : *pattern*, ..., d_n : *pattern*)

- A case term is written

```
{  p1   =>  t1
|  p2   =>  t2
          ⋮
|  pm   =>  tm
}  (t)
```

- A fold term is written

$$\boxed{\begin{array}{lllllll}
\texttt{\{|} & c_1 & : & & p_{11} & \texttt{=>} & t_{11} \\
 & & & \texttt{|} & p_{12} & \texttt{=>} & t_{12} \\
 & & & & & \vdots & \\
 & & & \texttt{|} & p_{1m} & \texttt{=>} & t_{1m} \\
 & \vdots & & & & & \\
\texttt{|} & c_n & : & & p_{n1} & \texttt{=>} & t_{n1} \\
 & & & \texttt{|} & p_{n2} & \texttt{=>} & t_{n2} \\
 & & & & & \vdots & \\
 & & & \texttt{|} & p_{nm} & \texttt{=>} & t_{nm} \\
\texttt{|\}} & (t) & & & & &
\end{array}}$$

- A map term is written

$$\boxed{\begin{array}{llll}
\texttt{L\{} & p_{11} & \texttt{=>} & t_{11} \\
\texttt{|} & p_{12} & \texttt{=>} & t_{12} \\
 & & \vdots & \\
\texttt{|} & p_{1m} & \texttt{=>} & t_{1m}\,\texttt{,} \\
\vdots & & & \texttt{,} \\
 & p_{n1} & \texttt{=>} & t_{n1} \\
\texttt{|} & p_{n2} & \texttt{=>} & t_{n2} \\
 & & \vdots & \\
\texttt{|} & p_{nm} & \texttt{=>} & t_{nm} \\
\texttt{\}} & (t) & &
\end{array}}$$

- An unfold term is written

$$\boxed{\begin{array}{lllllll}
\texttt{(|} & p_1 & \texttt{=>} & & d_1 & : & t_{11} \\
 & & & \texttt{|} & d_2 & : & t_{12} \\
 & & & & & \vdots & \\
 & & & \texttt{|} & d_n & : & t_{1n} \\
 & & \vdots & & & & \\
\texttt{|} & p_m & \texttt{=>} & & d_1 & : & t_{m1} \\
 & & & \texttt{|} & d_2 & : & t_{m2} \\
 & & & & & \vdots & \\
 & & & \texttt{|} & d_n & : & t_{mn} \\
\texttt{|)} & (t) & & & & &
\end{array}}$$

The function definition that corresponds to the mathematical description of a program type is:

$$\boxed{\begin{array}{lllll}
\texttt{def}\ \ \mathit{fun}\,\texttt{\{}\ f_1, \ldots, f_n\ \texttt{\}}\ \texttt{:}\ A\ \texttt{->}\ B\ \texttt{=} & & & & \\
 & & p_1 & \texttt{=>} & t_1 \\
 & \texttt{|} & p_2 & \texttt{=>} & t_2 \\
 & & & \vdots & \\
 & \texttt{|} & p_m & \texttt{=>} & t_m
\end{array}},$$

where both the parameter list (`{` $f_1, \ldots, f_n$ `}`) and the typing information (A `->` B) are optional.

An additional function definition syntax,

$$\boxed{\texttt{def}\ \ \mathit{fun}\ \texttt{\{}\ f_1, \ldots, f_n\ \texttt{\}}\ :\ A\ \texttt{->}\ B\ \texttt{=}\ \mathit{program}},$$

where *program* is

program $\rightarrow$

closed abstraction

| *fold*

| *unfold*

| *map*

is allowed to provide a shorthand for function definitions where patterns do not need to be explicitly defined. Again, both the parameter list and the typing information are optional.

Finally, the original function definition syntax,

$$\boxed{\texttt{def}\ \ \mathit{fun}\ \texttt{\{}\ f_1, \ldots, f_n\ \texttt{\}}\ \texttt{(}\ \mathit{variablebase}\ \texttt{)}\ \texttt{=}\ \mathit{term}}$$

was retained to provide backwards compatibility.

Chapter 7

Type Checking

Charity is a statically typed language; types cannot change during program execution. This means that all type checking can be done at the time of program compilation. Not only does this allow quicker execution of the actual program, it is also a good test of the programmer's implementation of the program specification. An incorrect implementation will not pass typechecking.

Since a **charity** program is simply a sequence of categorical combinators (see section 2.5), with specified domains and codomains, to do typechecking it is only necessary to ensure that the codomain of one combinator unifies with the domain of the next combinator in sequence; taking into account any restriction of type variables caused by previous unifications. In previous versions of **charity**, v3.11 written in SML and the Chirp interpreter written in C (Schroeder, 1993), typechecking was done at the combinator level.

For this project, however, we wish to do typechecking at the term logic level. There are four reasons for this.

- As will be seen in chapter 8, by assuming that the program is correctly typed, the function for translating pattern matching term logic to core term logic can be made less complex and more modular. This, in turn, will make it easier to show that the translation program is correct.

- By detecting any typechecking errors before the translation, optimization and conversion steps, compilation time will be saved (for a program with type errors).

- It should be possible to deliver much higher quality error messages from the term logic level, where the environment the error is detected in is the same as the environment the user was programming in.

- A version of **charity** with monads is currently being refined by Tom Fukushima. Typechecking is required at the term logic level for this version; in fact, Fukushima has already implemented a limited version of term logic typechecking for the core term logic.

Typing for the extended term logic was defined in section 6.2. To implement type checking we need three things.

- A unification algorithm.
- A type checker for a group of pattern phrases.
- Several functions to complete type checking for the various terms.

During type checking it will be important to track variables and make sure scoping rules are followed. Additionally, an explicit check must be made on patterns to make sure they do not contain two variables with the same name. This would imply equality and, in general, this cannot be computed in **charity**. Our assumption will be that this check has already been performed on any patterns that are being type checked.

Throughout the remainder of this chapter $'A$ will indicate a unique type variable. The functions **occurs_in** and **do_subs** will not be explicitly defined. Function **occurs_in** performs an occurs check for a type variable in a type. Function **do_subs** takes two arguments. One is a list of substitutions of the form $('A, type) :: S$. This indicates that type variable $'A$ is replaced by *type*. The second argument to **do_subs** is a structure which contains some number of type variables. Function **do_subs** replaces all type variables in the structure with their associated type from the substitution list.

7.1 The Unification Algorithm

The **unify** algorithm (see figure 7.1) works on two types, t_1 and t_2. It attempts to find a most general unification for the two types. If it finds one it is returned. If t_1 and t_2

- are both type variables such that t_1 and t_2 are the same then there are no substitutions,

- are such that t_i is a type variable, and t_i does not occur in the other type, t_j, then t_i goes out for t_j; otherwise it is a type error (infinite type),
- are such that neither is a type variable then they are of the form $Name(typelist)$. If $Name_1$ equals $Name_2$ then unify the *typelists* by calling **unify_typelists** (see figure 7.2); otherwise it is a type error (mismatched type Names).

The **unify_typelists** function (see figure 7.2) will unify each corresponding pair of types in the *typelists* and build a substitution list in the process. If the *typelists* are of unequal length then an error is raised. Whenever substitutions are found the substitution is made in the rest of the types in the *typelist*. This bounds the number of substitutions made by the number of type variables in the types being unified.

```
Function unify(t1, t2, S) =
   case (t1, t2) of
     ('A,'A)    => return S
     ('A,t2)    => if occurs_in('A, t2) then
                     raise infinite type error
                   else
                     return ('A, t2) :: S
     (t1,'A)    => if occurs_in('A, t1) then
                     raise infinite type error
                   else
                     return ('A, t1) :: S
     (Name1(typelist1), Name2(typelist2)) => if Name1 = Name2 then
                                               return unify_typelists(typelist1, typelist2, S)
                                             else
                                               raise mismatched types error
```

Figure 7.1: Function **unify**

```
Function unify_typelists([], [], S) = S
Function unify_typelists([], _, _) = type error (unequal type list lengths)
Function unify_typelists(_, [], _) = type error (unequal type list lengths)
Function unify_typelists(t1 :: t1s, t2 :: t2s, S) =
   let S' = unify(t1, t2, S)
   return unify_typelists(do_subs(t1s, S'), do_subs(t2s, S'), S')
```

Figure 7.2: Function **unify_typelists**

7.2 Typing Patterns

Patterns closely resemble terms. The difference is that patterns can introduce new variables while terms cannot. Thus, while typing patterns, one must build a list of variables that have been introduced along with their assigned types. Initially, each variable is given a unique type variable but, as the type checking progresses, the variable may have a more specific type assigned to it. If new variables are always added to the front of the variable list then it is a trivial matter to keep track of the scope of the variable. When a variable is encountered in a term its type can be determined by locating the first variable of the same name in the variables list. After every unification, the variable list must be updated with any substitutions found. This is done with the **do_subs** function.

We will now describe the **type_pattern** function (see figure 7.3). Let p be a pattern in the extended **charity** term logic and let $vars$ be a list of typed variables such that their scope includes p. Then the type of p and the specification of $vars$ can be determined as follows. If p is

- the null pattern, $()$, then p has type 1 and $vars$ is unchanged.
- a variable, v, then v is assigned $vtype$, where $vtype$ is a type variable new to both p and $vars$. $vars$ becomes $(v, vtype) :: vars$.
- a pair, (p_1, p_2), then the type of p is the product of the types of p_1 and p_2. $vars$ has any new variables in p_1 and p_2 added to it.
- a constructor pattern, $c_i(p_i)$ with domain $E_i(A, L(A))$ and codomain $L(A)$, then the type of p_i must unify with $E_i(A, L(A))$. Any type parameters in $E_i(A, L(A))$ are assigned new type variables. After unification, $L(A)$ is updated from the unifier substitution list and is returned as the type of the pattern. $vars$ has had any new variables in p_i added to it.
- a destructor pattern, $(d_1 : p_1, \ldots, d_n : p_n)$ with domain $R(A)$ and codomain $F_i(A, R(A))$, then the type of any p_i must unify with $F_i(A, R(A))$. Any type parameters in $F_i(A, R(A))$ are assigned new type variables. After all unifications have taken place, $R(A)$ is updated from the unifier substitution list and is returned as the type of the

pattern. $vars$ has had any new variables in p_i added to it. Much of this processing is done in functions **type_destr_patt** and **type_destr** (see figures 7.4 and 7.5).

Function **type_pattern**$(p, vars)$ =
  case p of
  $()$ => $(1, vars)$
  v => let $vartype$ = **getnewvartype**()
    return $(vartype, (v, vartype) :: vars)$
  (p_1, p_2) => let $(tp_1, vars')$ = **type_pattern**$(p_1, vars)$
    let $(tp_2, vars'')$ = **type_pattern**$(p_2, vars')$
    return $(\times(\mathrm{tp}_1,\ \mathrm{tp}_2),\ vars'')$
  $c_i(p_i)$ => let $(tp, vars')$ = **type_pattern**$(p, vars)$
    let S = **unify**$(tp,\ E_i(A, L(A)),\ [])$
    return (**do_subs**$(L(A), S)$, **do_subs**$(vars', S)$)
  $(d_1 : p_1, \ldots, d_n : p_n)$ => let $(vars', R(A)s)$ = **type_destr_patt**$((d_1 : p_1, \ldots, d_n : p_n),\ vars, [])$
    let S = **unifylist**$(R(A)s)$
    return (**do_subs**(head$(R(A)s, S)$), **do_subs**$(vars', S)$)

Figure 7.3: Function **type_pattern**

Function **type_destr_patt**$([], vars, R(A)s) = (vars, R(A)s)$

Function **type_destr_patt**$((d_i : p_i, \ldots, d_n : p_n), vars, R(A)s)$ =

  let $(tR(A), vars')$ = **type_destr**$(R(A), F_i(A, R(A)), p_i, vars)$
  return **type_destr_patt**$((d_{i+1} : p_{i+1}, \ldots, d_n : p_n),\ vars', R(A) :: R(A)s)$

Figure 7.4: Function **type_destr_patt**

Function **type_destr**$(R(A), F_i(A, R(A)), p_i, vars)$ =

  let $(tp_i, vars')$ = **type_pattern**$(p_i, vars)$
  let S = **unify**$(tp_i,\ F_i(A, R(A)),\ [])$
  return (**do_subs**$(R(A), S)$, **do_subs**$(vars', S)$)

Figure 7.5: Function **type_destr**

7.3 Typing Terms

The function for typing terms is very similar to function **type_pattern**. However, if a new variable is found in a term, it is an unbound variable which is a type error.

Central to the typing of the pattern matched terms, the case, the fold, the unfold and the map, is the **get_group_type** function (see figure 7.6). It and its supporting functions, **type_phrases** (figure 7.7), **type_a_phrase** (figure 7.8), **unify_all_patts** (figure 7.9), and **unify_all_terms** (figure 7.10), type a group of phrases. The function **get_group_type** returns the type of the patterns and the terms in the group. It also returns an updated *vars* list. This list does *not* include any variables found in the patterns of the group. The **type_term** function up to the case term is given in figure 7.11. The typing procedures for the fold, map, unfold, and record terms can be derived using the typing definitions given in section 6.2 and the subsidiary functions already developed.

Function $\textbf{get\_group\_type}\left(\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, vars\right) =$

$\quad$ let $(tpatts, tterms, vars') = \textbf{type\_phrases}\left(\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, 'A, 'B, vars\right)$
$\quad$ let $(tpatts', S) = \textbf{unify\_all\_patts}(tpatts, 'C, [])$
$\quad$ let $(tterms', S') = \textbf{unify\_all\_terms}(tterms, 'D, S)$
$\quad$ return $(tpatts', tterms', \textbf{do\_subs}(vars', S'))$

Figure 7.6: Function **get_group_type**

Function $\textbf{type\_phrases}([], tpatts, tterms, vars) = (tpatts, tterms, vars)$

Function $\textbf{type\_phrases}\left(\begin{array}{ccc} p_i & \mapsto & t_i \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, tpatts, tterms, vars\right) =$
$\quad$ let $(tp_i, tt_i, vars') = \textbf{type\_a\_phrase}(p_i \mapsto t_i,\ vars)$
$\quad$ return $\textbf{type\_phrases}\left(\begin{array}{ccc} p_{i+1} & \mapsto & t_{i+1} \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, tp_i :: tpatts, tt_i :: tterms, vars'\right)$

Figure 7.7: Function **type_phrases**

Function **type_a_phrase**($p_i \mapsto t_i$, $vars$) =

let $(tp_i, vars')$ = **type_pattern**$(p_i, vars)$
let $(tt_i, vars'')$ = **type_term**$(t_i, vars')$
return $(tp_i, tt_i,$ **update**$(vars, vars''))$

Figure 7.8: Function **type_a_phrase**

Function **unify_all_patts**$([], type, S) = (type, S)$

Function **unify_all_patts**$(tp :: tpatts, type, S)$ =
let S' = **unify**$(type, tp, S)$
return **unify_all_patts**$(tpatts,$ **do_subs**$(type, S'), S')$

Figure 7.9: Function **unify_all_patts**

Function **unify_all_terms**$([], type, S) = (type, S)$

Function **unify_all_terms**$(tt :: tterms, type, S)$ =
let S' = **unify**$(type,$ **do_subs**$(tt, S), S)$
return **unify_all_terms**$(tterms,$ **do_subs**$(type, S'), S')$

Figure 7.10: Function **unify_all_terms**

Function **type_term**$(t, vars)$ =
  case t of
    $()$ => $(1, vars)$
    $\mathrm{p}_0(t')$ => let $(tt, vars') = \textbf{type\_term}(t', vars)$
        return $(\textbf{do\_subs}('A, \textbf{unify}(\times('A,'B), tt, [])), vars')$
    $\mathrm{p}_1(t')$ => let $(tt, vars') = \textbf{type\_term}(t', vars)$
        return $(\textbf{do\_subs}('B, \textbf{unify}(\times('A,'B), tt, [])), vars')$
    (t_1, t_2) => let $(X, vars') = \textbf{type\_term}(t_1, vars)$
        let $(Y, vars'') = \textbf{type\_term}(t_2, vars')$
        return $(\times(X, Y), vars'')$
    v => if member$(v, vars)$ then
          return $(vtype, vars)$
        else
          raise type error, unbound variable
    $c_i(t_i)$ => let $(tt_i, vars')) = \textbf{term\_type}(t_i, vars)$
        let $S = \textbf{unify}(tt_i, E_i(A, L(A)), [])$
        return $(\textbf{do\_subs}(L(A), S), \textbf{do\_subs}(vars', S))$
    $\left\{\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}\right\}(t')$ => let $(tp, tt, vars') = \textbf{get\_group\_type}\left(\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, vars\right)$
        let $(tt', vars'') = \textbf{type\_term}(t', vars')$
        let $S = \textbf{unify}(tt', tp, [])$
        return $(\textbf{do\_subs}(tt', S), \textbf{do\_subs}(vars'', S))$

Figure 7.11: Function **type_term**

Chapter 8

Translating The Pattern Matching Term Logic

The term logic must be converted to categorical combinators before it can be evaluated by the abstract machine. A conversion has been given in *About Charity* (Cockett & Fukushima, 1992) but it is for the core term logic only. Therefore the pattern matching term logic must be translated to the core term logic before it can be converted to categorical combinators.

Translation algorithms already exist for translating patterns into nested case statements. One is given by Wadler (Bird & Wadler, 1988), and a similar one is given by Cockett and Simpson (Cockett & Simpson, 1993). The chief difference between the two algorithms are in the handling of the patterns beginning with a mixture of variables and constructors. Wadler's algorithm is generally more circumspect in generating nested case statements, but it pays a price in that it is more difficult to understand. Cockett and Simpson's algorithm generates more case statements but is simpler. Most of these extra case expressions can be removed by optimization using decision tree reductions (Cockett & Simpson, 1993; Cockett & Herrera, 1990).

However, neither algorithm handles pairs or destructor patterns. These are handled in the algorithm presented here by deferring processing of the later components of the pair or destructor pattern. The algorithm presented here most closely resembles Cockett and Simpson's algorithm.

8.1 Explanation of the Translation Algorithm

When translating a group of pattern phrases there are three possibilities to consider. The patterns could

- contain a coinductive datatype or pair, or
- contain an inductive datatype, or
- be all basic types (variables or nulls).

In the first case we want to translate the term according to the patterns associated with each destructor in the term. To do this we must delay the translation of the patterns associated with the other destructors until later. This is accomplished through the means of a stack, S, onto which the relevant information is pushed. S is initially empty when the translation begins. When a destructor pattern[1] is encountered a tuple consisting of

- the instantiating term for the destructor pattern,
- a list of destructors for the pattern type, and
- the current group of patterns

is pushed onto S. This processing, which also determines the group of patterns for the next call to the translation algorithm, is done in function **dodestr** (see figure 8.1).

In the second case, a core term logic case expression is constructed using the constructors of the datatype $L(A)$. The term corresponding to each constructor, c_i, is a translation of just those pattern phrases whose pattern is either of the form $c_i(p_i)$ or is a variable. The pattern groups in the items on the stack, S, are pruned so they contain only the corresponding patterns. This processing is done in function **doconstr** (see figure 8.3). This step has the effect of splitting the translation into several branches, each of which is dealing with some subgroup of the original phrases. If there are no phrases in a particular subgroup then it indicates the initial group of patterns was not complete. This is an error and will terminate the translation process.

[1] A pattern (p_1,p_2) of type $A \times B$ can be thought of, in this context, as a destructor pattern with the form $(\mathrm{p}_0 : p_1, \mathrm{p}_1 : p_2)$.

In the last case, the patterns are variables or nulls. If the stack is empty then we are done. If there are more than one pattern phrases left at this point then the initial set of patterns was overlapping; although not necessarily completely overlapping. The uppermost pattern phrase is returned.

If S is not empty an item is popped from the stack. The pattern phrases to be used in the next call to the translation algorithm are determined using function **getdp** (see figure 8.2).

Function **getdp** returns a pattern phrase. If the pattern passed to **getdp** is a destructor pattern then a pattern phrase consisting of the pattern for the jth destructor mapped to an abstraction is returned. The abstraction consists of the original pattern phrase with the jth-destructed coinductive data type applied to it.

Before presenting the translation algorithm itself it is worth noting a few things. First, the order of the patterns is important. As described in section 4.1.1, overlapping patterns are allowed but, in the case of an input matching more than one pattern, it is the pattern that occurs first in the group that is chosen. Thus, in all cases, it should be assumed that processing does not alter the order of the pattern phrases.

On the other hand, the order the destructors are processed inside a destructor pattern is not important; the results will be the same. Indeed, it is not even required that all destructors be present in the pattern; a missing destructor can be assigned a *don't care* variable without loss of meaning.

The reader should also assume that any variable of the form x_i introduced into the translated term is new to that term.

8.2 Translation Algorithm for Patterns

Let *ptype* be the most general type of the patterns in

$$\begin{array}{lcl} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array} .$$

Then,

$$\left[\!\!\left[\left(\begin{array}{lcl} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array} , S \right) \right]\!\!\right]$$

is a translation of this group of phrases where,

- if $ptype = A \times B$ or $R(A)$ then

$$\boxed{x_k \mapsto \left\{ \left[\!\left[\text{dodestr} \left(x_k, [d_1, \ldots, d_n], \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, S \right) \right]\!\right] \right\} (d_1(x_k))} \quad ;$$

- if $ptype = L(A)$ then

$$\boxed{\begin{array}{lcl} c_1(x_k) & \mapsto & \left\{ \left[\!\left[\text{doconstr} \left(c_1, \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, S \right) \right]\!\right] \right\} (x_k) \\ \vdots & & \\ c_n(x_k) & \mapsto & \left\{ \left[\!\left[\text{doconstr} \left(c_n, \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, S \right) \right]\!\right] \right\} (x_k) \end{array}} \quad ;$$

- if $ptype = A$ or 1 then

$$\boxed{\begin{array}{l} \text{if } S \text{ is empty then} \\ \quad \text{if } (m > 1) \text{ then} \\ \quad\quad \text{print "Warning: There are overlapping patterns."} \\ \quad p_1 \mapsto t_1 \\ \text{else} \\ \quad \text{let } \left(x_k, [d_i, d_j, \ldots, d_n], \begin{array}{c} p_{1_S} \\ \vdots \\ p_{m_S} \end{array} \right) :: S' = S \\ \quad \text{let } S'' = S' \text{ if } i = n; \left(x_k, [d_j, \ldots, d_n], \begin{array}{c} p_{1_S} \\ \vdots \\ p_{m_S} \end{array} \right) \text{ otherwise} \\ \quad x_{k+1} \mapsto \left\{ \left[\!\left[\left(\begin{array}{c} \text{getdp}(d_j, p_{1_S}, x_{k+1}, p_1 \mapsto t_1) \\ \vdots \\ \text{getdp}(d_j, p_{m_S}, x_{k+1}, p_m \mapsto t_m) \end{array}, S'' \right) \right]\!\right] \right\} (d_i(x_k)) \end{array}} \quad .$$

8.2.1 Case, Fold, Map and Unfold Patterns

The translation of the various pattern matched terms to core **charity** term logic is straightforward given the translation function for complete pattern phrases described above.

- A pattern matched case term

$$\left\{\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}\right\}(t)$$

  is translated by

$$\left\{\left[\!\!\left[\begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, []\right]\!\!\right]\right\}(t).$$

- A pattern matched fold term

$$\left\{\left|\begin{array}{l} c_1 : \left|\begin{array}{ccc} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1} \end{array}\right. \\ \vdots \\ c_n : \left|\begin{array}{ccc} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n} \end{array}\right. \end{array}\right|\right\}(t)$$

  is translated by

$$\left\{\left|\begin{array}{l} c_1 : \left|\left[\!\!\left[\begin{array}{ccc} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1} \end{array}, []\right]\!\!\right]\right. \\ \vdots \\ c_n : \left|\left[\!\!\left[\begin{array}{ccc} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n} \end{array}, []\right]\!\!\right]\right. \end{array}\right|\right\}(t).$$

- A pattern matched map term

$$L\left\{\begin{array}{l} \left|\begin{array}{lcl} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1} \end{array}\right. \\ \vdots \\ \left|\begin{array}{lcl} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n} \end{array}\right. \end{array}\right\}(t)$$

is translated by

$$L\left\{\begin{array}{l} \left|\left[\!\!\left[\begin{array}{lcl} p_{11} & \mapsto & t_{11} \\ & \vdots & \\ p_{1m_1} & \mapsto & t_{1m_1} \end{array}, []\right]\!\!\right]\right. \\ \vdots \\ \left|\left[\!\!\left[\begin{array}{lcl} p_{n1} & \mapsto & t_{n1} \\ & \vdots & \\ p_{nm_n} & \mapsto & t_{nm_n} \end{array}, []\right]\!\!\right]\right. \end{array}\right\}(t).$$

- A pattern matched unfold term

$$\left(\!\left|\begin{array}{lcl} p_1 & \mapsto & \left|\begin{array}{lcl} d_1 & : & t_{11} \\ & \vdots & \\ d_n & : & t_{1n} \end{array}\right. \\ & \vdots & \\ p_m & \mapsto & \left|\begin{array}{lcl} d_1 & : & t_{m1} \\ & \vdots & \\ d_n & : & t_{mn} \end{array}\right. \end{array}\right|\!\right)(t).$$

is first rearranged by introducing an abstraction and gathering the patterns up under each destructor as follows:

$$\left(\left| x_1 \mapsto \begin{array}{ccc} d_1 & : & \left\{\begin{array}{ccc} p_1 & \mapsto & t_{11} \\ & \vdots & \\ p_m & \mapsto & t_{m1} \end{array}\right\}(x_1) \\ \vdots & & \\ d_n & : & \left\{\begin{array}{ccc} p_1 & \mapsto & t_{1n} \\ & \vdots & \\ p_m & \mapsto & t_{mn} \end{array}\right\}(x_1) \end{array} \right|\right)(t);$$

then the translation is performed like this

$$\left(\left| x_1 \mapsto \begin{array}{ccc} d_1 & : & \left\{\left[\!\left[\begin{array}{ccc} p_1 & \mapsto & t_{11} \\ & \vdots & \\ p_m & \mapsto & t_{m1} \end{array}, []\right]\!\right]\right\}(x_1) \\ \vdots & & \\ d_n & : & \left\{\left[\!\left[\begin{array}{ccc} p_1 & \mapsto & t_{1n} \\ & \vdots & \\ p_m & \mapsto & t_{mn} \end{array}, []\right]\!\right]\right\}(x_1) \end{array} \right|\right)(t).$$

$$\text{Function } \mathbf{dodestr}\left(x_k, [d_1, \ldots, d_n], \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, S \right) =$$

$$\text{let newphrases} = \begin{array}{c} \text{getdp}(d_1, p_1, x_k, p_1 \mapsto t_1) \\ \vdots \\ \text{getdp}(d_1, p_m, x_k, p_m \mapsto t_m) \end{array}$$

$$\text{let } S' = \left(x_k, [d_2, \ldots, d_n], \begin{array}{c} p_1 \\ \vdots \\ p_m \end{array} \right) :: S \text{ if } n > 1;\ S \text{ otherwise}$$

return (newphrases, S')

Figure 8.1: Function **dodestr**

Function $\mathbf{getdp}(d_j, p_{i_S}, x, p_i \mapsto t_i) =$
if p_{i_S} is a destructor pattern of the form $(d_1 : p_{1S_{d_1}} \ldots d_n : p_{nS_{d_n}})$ then
return $p_{iS_{d_j}} \mapsto \{p_i \mapsto t_i\}(x)$
else
return $x_{k+2} \mapsto \{p_i \mapsto t_i\}(x)$

Figure 8.2: Function **getdp**

Function **doconstr** $\left(c_i(x_k), \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, S \right) =$

let f $c_i(p_{c_i}) \mapsto t_j$ = TRUE
  f $v \mapsto t_j$ = TRUE
  f _ = FALSE

let g $c_i(p_{c_i}) \mapsto t_j = p_{c_i} \mapsto t_j$
  g $v \mapsto t_j = x_{k+j} \mapsto \{v \mapsto t_j\}(c_i(x_k))$

let newphrases = map g $\left(\text{filter f} \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array} \right)$

let h (term, destrs, phrases) = $\left(\text{term, destrs, h}' \left(\text{phrases}, \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, [] \right) \right)$

let h$'$([], _, Ps) = rev(Ps)

$$\text{h}' \left(\text{p::ps}, \begin{array}{ccc} p_j & \mapsto & t_j \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, Ps \right) = \begin{array}{l} \text{if (f } p_j \mapsto t_j \text{) then} \\ \quad \text{h}' \left(ps, \begin{array}{ccc} p_{j+1} & \mapsto & t_{j+1} \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, p :: Ps \right) \\ \text{else} \\ \quad \text{h}' \left(ps, \begin{array}{ccc} p_{j+1} & \mapsto & t_{j+1} \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}, Ps \right) \end{array}$$

let S$'$ = map h S

if newphrases is empty then
  raise "Error: Pattern matching not exhaustive"
else
  return (newphrases, Ps)

Figure 8.3: Function **doconstr**

Chapter 9

Conclusion

This report has given definitions for patterns in **charity** and redefined the **charity** term logic so that it is suitable for pattern matching. A programming syntax was developed from this. In addition, an algorithm was given for translating the extended term logic to the core term logic. An algorithm for typechecking the extended term logic was also described.

As of this writing, the programming syntax has been fully implemented into v3.11 of the **charity** interpreter. A first attempt at implementing the translation algorithm must be redone, but this should be completed by April 15th.

Because of the constraints of the CPSC 502 course, this paper had to be written before the pattern matching was fully implemented. It still remains to implement the term logic type checker. In addition, optimization of the translated core term logic needs to be done. This will involve learning the concepts of decision tree optimization and implementing an optimizer in the compiler.

In the longer term, the pattern matching compiler and monad compiler need to be integrated. It is the intention of the author to complete this project, in its entirety, as part of a Master's programme beginning in September, 1994.

Appendix A

Simple Boolean Parser

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

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

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

data tokens -> C =
         T   : 1 -> C
        |F   : 1 -> C
        |AND: 1 -> C
        |OR  : 1 -> C
        |NEG: 1 -> C.

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

def length(lst) = {| nil  : ()    => zero
                  | cons: (_,x) => succ(x)
                  |}(lst).

def dostart(expr) =
  {nil()           => (ff, nil)
  |cons(tk1,expr') => {T()   => (ss(true)  , expr')
                      |F()   => (ss(false), expr')
                      |NEG() => {nil() => (ff, nil)
                                |cons(tk2,expr'') => {T() => (ss(false),expr'')
                                                     |F() => (ss(true),expr'')
                                                     | _  => (ff, nil)
                                                     }(tk2)
                                }(expr')
                      | _    => (ff, nil)
                      }(tk1)
  }(expr).
```

```
def doeval(r, expr) =
  {nil()            => (r,nil)
  |cons(tk1,expr') =>
     {AND() => {nil()             => (ff, nil)
               |cons(tk2,expr'') =>
                  {T()    => {ff()   => (ff, nil)
                             |ss(b) => {true()   => (ss(true) ,expr'')
                                        |false() => (ss(false),expr'')
                                        }(b)
                             }(r)
                  |F()    => (ss(false),expr'')
                  |NEG() => {nil()              => (ff, nil)
                             |cons(tk3,expr''') =>
                                {T() => (ss(false),expr''')
                                |F() => {ff()   => (ff, nil)
                                        |ss(b) =>
                                            {true() => (ss(true),expr''')
                                           |false() => (ss(false),expr''')
                                           }(b)
                                        }(r)
                                |_   => (ff, nil)
                                }(tk3)
                             }(expr'')
                  | _    => (ff, nil)
                  }(tk2)
                }(expr')
     |OR()  => {nil()             => (ff, nil)
               |cons(tk2,expr'') =>
                  {T()    => (ss(true),expr'')
                  |F()    => {ff()   => (ff, nil)
                             |ss(b) => {true()   => (ss(true) ,expr'')
                                        |false() => (ss(false),expr'')
                                        }(b)
                             }(r)
                  |NEG() => {nil()              => (ff, nil)
                             |cons(tk3,expr''') =>
                                {T() => {ff()   => (ff, nil)
                                        |ss(b) =>
                                            {true() => (ss(true),expr''')
                                           |false() => (ss(false),expr''')
                                           }(b)
                                        }(r)
                                |F() => (ss(true),expr''')
                                |_   => (ff, nil)
                                }(tk3)
                             }(expr'')
                  | _    => (ff, nil)
                  }(tk2)
                }(expr')
     | _    =>        (ff, nil)
     }(tk1)
  }(expr).

def parse(expr) =
  p0({| zero: ()          => dostart(expr)
      | succ: (r, expr') => {ff      => (ff,nil)
                              |ss(b) => doeval(ss(b), expr')
                            }(r)
      |}(length(expr))).
```

Appendix B

Variable Bases

Patterns, in the extended term logic, replace the variable bases of the core term logic. Variable bases were defined in the core term logic (Cockett & Fukushima, 1992) as follows:

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

Appendix C

Core Term Logic

A single variable base in the core term logic can be replaced by one or more patterns in the extended term logic. Thus, terms in the core term logic that used variable bases were changed in the extended term logic.

C.1 Term Logic

Following is a list of the core term logic terms that were modified:

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

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

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

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

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

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

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

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

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

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

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

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

C.2 Abstracted Maps

A program is not a term but an **abstracted map** this is a pair

$$v \mapsto t$$

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

Appendix D

Original Charity Grammar

This is the grammar as it was in Version 3.11 of Charity. This grammar was subsequently modified to introduce pattern matching capabilities.

```
def →
    : ID parms patty “=” expr

parms →
    : “{” id_list “}”
    | “{” “}”
    |

id_list →
    : ID “,” id_list
    | ID

patty →
    : pattx
    |

pattx →
    : “(” patt “,” patt “)”
    | “(” patt “)”
    | “(” “)”

patt →
    : “(” patt “,” patt “)”
    | “(” patt “)”
    | “(” “)”
    | ID
    | “_”

exprx →
    : “(” expr “,” expr “)”
    | “(” expr “)”
    | “(” “)”

expry →
    : exprx
    |

expr →
    : “(” “)”
    | “(” expr “)”
    | “(” expr “,” expr “)”
    | “{” abs “}” exprx
    | “{” casee “}” exprx
    | “{|” fold “|}” exprx
    | “(|” unfold “|)” exprx
    | “(” record “)”
    | ID expry
    | ID “{” strengths “}” exprx
    | “[“ expr_list “]”
    | IF expr THEN expr ELSE expr
    | STRING
    | INTNEG
    | INTPOS

expr_list →
    : expr
    | expr “,” expr_list
    |

casee →
    : cases
    | cases “|” “_” “=>” expr

cases →
    : cse
    | cases “|” cse

cse →
    : ID pattx “=>” expr
    | ID “=>” expr

fold →
    : folds
    | folds “|” “_” “=>” expr

folds →
    : fld
    | folds “|” fld

fld →
    : ID “:” patt “=>” expr

unfold →
    : patt “=>” unfolds
```

unfolds →
: unfld
| unfolds "|" unfld

unfld →
: **ID** ":" expr

record →
: records

records →
: recrd
| records "," recrd

recrd →
: **ID** ":" expr
| opt

strengths →
: abs
| abs "," strengths

absL →
: abs
| abs "," absL

abs →
: patt "=>" expr
| comb_abs

opt →
: ()
| "," ()

comb_abs →
: **ID**
| comb_abs ";" **ID**
| **ID** "{" absL "}"
| comb_abs ";" **ID** "{" absL "}"

Appendix E

Extended Charity Grammar

This is the grammar for Charity as modified from the grammar in appendix reforiginal_grammar to add pattern matching capabilities. Definitions that were added or altered are indicated by †.

```
def† →
    : ID parms "=" expr
    | ID parms vbasex "=" expr
    | ID parms typing "=" defbody

parms →
    : "{" id_list "}"
    | "{" "}"
    |

typing† →
    : ":" idtype "->" idtype
    |

idtype† →
    : "(" idtype ")"
    | ID "(" type_list ")"
    | ID

idtype_list† →
    : idtype "," idtype_list
    | idtype

id_list →
    : ID "," id_list
    | ID

vbasex† →
    : "(" vbase ")"
    | "(" ")"

vbase† →
    : ID
    | vbase "," vbase
    | "(" vbase ")"
    | "(" ")"
    | "_"

def_body† →
    : "{" cases "}"
    | "{|" fld "|}"
    | "(|" unfld "|)"
    | ID "{" strengths "}"
    | cases

(* A pattern is one of the following: *)
(* ID and UNDERSCORE are syntax extensions *)
patt† →
    : pattx
    | ID
    | ID pattx
    | "_"

pattx† →
    : "(" ")"
    | "(" patt ")"
    | "(" patt "," patt ")"
    | "(" pattd ")"

pattd† →
    : ID ":" patt
    | pattd "," ID ":" patt

exprx →
    : "(" expr "," expr ")"
    | "(" expr ")"
    | "(" ")"

expry →
    : exprx
    |

expr† →
    : "(" ")"
    | "(" expr ")"
```

| "(" expr "," expr ")"
| "{" abs "}" exprx
| "{" cases "}" exprx
| "{|" fold "|}" exprx
| "(|" unfold "|)" exprx
| "(" record ")"
| **ID** expry
| **ID** "{" strengths "}" exprx
| "[" expr_list "]"
| **IF** expr **THEN** expr **ELSE** expr
| **STRING**
| **INTNEG**
| **INTPOS**

expr_list →
: expr
| expr "," expr_list
|

cases† →
: casephrase
| cases "|" casephrase

casephrase† →
: patt "=>" expr

fld† →
: fldsntnce

fldsntnce† →
: **ID** ":" patt "=>" expr
| **ID** ":" patt "=>" expr "|" fldphrase

fldphrase† →
: fldsntnce
| patt "=>" expr
| patt "=>" expr "|" fldphrase

unfld† →
: unfldsntnce

unfldsntnce† →
: patt "=>" **ID** ":" expr
| patt "=>" **ID** ":" expr "|" unfldphrase

unfldphrase† →
: unfldsntnce
| **ID** ":" expr
| **ID** ":" expr "|" unfldphrase

record →
: records

records →
: recrd
| records "," recrd

recrd →
: **ID** ":" expr
| opt

strengths† →
: abs_comb
| map

abs_comb† →
: abs
| abs "," abs_comb

map† →
: cases
| cases "," map

abs† →
: abs_comb

comb_abs →
: **ID**
| comb_abs ";" **ID**
| **ID** "{" absL "}"
| comb_abs ";" **ID** "{" absL "}"

absL →
: abs
| abs "," absL

Bibliography

Backus, J. (1978). Can Programming Be Liberated From The von Neumann Style? A Functional Style And Its Algebra Of Programs. *Communications of the ACM*, *21*(8), 613–641.

Bird, R., & Wadler, P. (1988). *Introduction to Functional Programming*. Prentice-Hall International. From the Prentice-Hall International Series In Computer Science.

Cockett, J. R. B., & Herrera, J. A. (1990). Decision Tree Reduction. *Journal of the Association For Computing Machinery*, *37*(4), 815–842.

Cockett, R. (1993a). Charitable Thoughts.. University of Calgary CPSC 501 class notes in draft form.

Cockett, R. (1993b). Developing An Interpreter.. University of Calgary CPSC 501 class notes.

Cockett, R., & Fukushima, T. (1992). About Charity. Tech. rep. #92/480/18, University of Calgary. Research report.

Cockett, R., & Simpson, T. (1993). More On Compiling Pattern Matched Definitions. Unpublished.

Fukushima, T. (1991). Draft: Charity User Manual. Draft copy.

Holyer, I. (1991). *Functional Programming With Miranda*. Pitman Publishing, 128 Long Acre, London WC2E 9AN.

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, *32*(2), 98–107.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International. From the Prentice-Hall International Series In Computer Science.

Schroeder, M. (1993). A Front End For Charity. University of Calgary CPSC 502 project paper.

Tarditi, D. R., & Appel, A. W. (1991). ML-Yacc, version 2.1 Documentation for Release Version. Tech. rep., Carnegie Mellon and Princeton. A user's manual.

Walters, R. F. C. (1991). *Categories and Computer Science*. Cambridge University Press. Number 28 in the series: Cambridge Computer Science Texts.

Yee, D. B. (1993). The CHARM Project: A Back End To The Charity Interpreter. University of Calgary CPSC 502 project paper.