DRAFT

Charity User Manual

for Version 1.00 β

Tom Fukushima
and
Charles Tuckey

Department of Computer Science
University of Calgary
Calgary, Alberta T2N 1N4
CANADA
email: thf | tuckey@cpsc.ucalgary.ca

January 19, 1996

Contents

1 History

The language **Charity** was initially conceived and designed by Robin Cockett while he was a research fellow with the Sydney category seminar at Macquarie University from January 1990 to June 1991. Tom Fukushima wrote the first implementation of **charity** in SML during 1991 under the guidance of Robin Cockett. Charles Tuckey added pattern matching to a later incarnation of this implementation in 1994.

The current version of **charity**, α0.97i , that is covered in this manual was written almost entirely during the month of May, 1995 by Barry Yee, Charles Tuckey and Peter Vesely. It is a C implementation. Vesely wrote the typechecking module. Tuckey wrote the pattern matching and symbol table modules. Yee wrote the core term logic translator, the combinator compiler and the machine. The parser and lexer were written by Yee and Tuckey using a grammar essentially the same as the one used in Fukushima's SML version of **charity**.

Currently, Marc Schroeder and Robin Cockett are working on implementing higher order types in **charity**.

2 Introduction

Charity is an interactive environment useful for defining and executing **charity** programs. This document describes the commands available to manipulate, use and query the environment. We introduce the system through an example session in section ??. Section ?? is a long section that describes the **charity** system in some detail. It contains various subsections that explain the use of datatypes, functions and commands in **charity**. Finally, the grammar for **charity** is given in appendix ??.

The **charity** system is available via anonymous ftp from `cpsc.ucalgary.ca` in the pub/charity directory. It is also available on the WWW at `http://www.cpsc.ucalgary.ca/projects/charity/home.html`. Please send any inquiries, comments and bugs to `charity@cpsc.ucalgary.ca`.

3 Example session

We give a taste of the **charity** system through an extended example in which operations on a binary tree (btree) and supporting datatypes are performed.

After **charity** is started it will initially display:

```
The Charity Interpreter, Version 1.00 (beta)
      Charity Development Group - Dec 1995

Charity>>
```

The `Charity>>` prompt is displayed whenever the system is ready for the next command.

The first command we want is one that will allow us to construct natural numbers. For this we use the `data` command:

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

We use this datatype to represent the natural number 0:

```
Charity>> zero.
zero : nat
```

and the number 3:

```
Charity>> succ(succ(succ(zero))).
succ(succ(succ(zero))) : nat
```

A more complicated datatype can be used to represent btrees:

```
Charity>> data btree(A) -> C =
      leaf : 1 -> C
    | node : A * (C * C) -> C.
Datatype added: btree
```

which says that `btree` is a type that has two constructors: `leaf` and `node`. The `leaf` is a constructor which takes no arguments and returns something of type `btree(A)`. This can be obtained from the `leaf` phrase by replacing occurrences of `C` with `btree(A)`. Similarily, the type of `node` can be read as `node` takes an argument of type `A` and a pair of `btree(A)`s and returns something of type `btree(A)`. All commands are terminated by a `.` (period).

We can now create some `btrees`:

```
Charity>> leaf().
leaf : btree(A)
Charity>> node(succ(zero),(leaf,leaf)).
node(succ(zero), (leaf, leaf)) : btree(nat)
```

The first `btree` is just a `leaf` with no `node`s. Since there are no `node`s the system does not know what type of data the nodes can contain so it assigns it a most general type `A`. When the command is an expression, the system executes the expression and displays the result. The second `btree` is a node with a nat value whose subtrees are `leaf`s.

If we want to assign a btree to a name we can use the `def` command:

```
Charity>> def one = () => node(succ(zero),(leaf,leaf)).
Function added:  one : 1 -> btree(nat)
Charity>> one.
node(succ(zero), (leaf, leaf)) : btree(nat)
```

As you can see, subsequent uses of the function name `one` will be equivalent to using `node(succ(zero),(leaf,leaf))`. Since function `one` takes no arguments we use `()`, which has type `1`, in the definition. When using a function that takes no arguments it is not necessary to explicitly use `()` in the function call.

To create a tree with a root `node` containing `succ(succ(zero))` and `succ(zero)` as the subtrees we could enter:

```
Charity>> def two11 = () => node(succ(succ(zero)), (one, one)).
Function added:  two11 : 1 -> btree(nat)
Charity>> two11.
node(succ(succ(zero)), (node(succ(zero), (leaf, leaf)), node(succ(zero), (leaf, leaf)))) : btree(nat)
```

Supplied with every datatype declaration are three functions. With the left datatypes (typename is to the left of the arrow in the `data` declaration) we get a **fold**, **case** and **map** function for free. A **fold** function replaces the constructors of the datatype by functions and evaluates the resulting expression.

For example, to add two natural numbers together we can define a function that uses a fold:

```
Charity>> def addNat = (x,y) =>
                {| zero : () => y
                 | succ : y' => succ(y')
                 |}(x).
Function added:  addNat : nat * nat -> nat
```

Now we can use the addNat function in another fold that adds up all of the values in the nodes of a tree:

```
Charity>> def sumTree = tree =>
{| leaf : () => zero
                 | node : (num, (LeftT, RightT)) =>
 addNat(num, addNat(LeftT, RightT))
                 |}(tree).
Function added:  sumTree : btree(nat) -> nat
Charity>> sumTree(leaf).
zero : nat
Charity>> sumTree(two11).
succ(succ(succ(succ(zero)))) : nat
```

The **def** command says that **sumTree** takes one argument called **tree**. The **tree** is applied to a fold operation for **btree** which says: recurse over the **btree** returning **zero** for any subtree which is a **leaf** and the sum of the subtrees and the value of the node for any node. Expressed using general recursion, this could be written:

$$
\begin{array}{l}
\textit{sumTree(tree)} = \\
\quad \textit{case tree of} \\
\qquad \textit{leaf () => zero} \\
\quad\ \mid \textit{node (num,(LeftT,RightT)) => addNat(num,addNat(sumTree(LeftT),sumTree(RightT)))}
\end{array}
$$

The **case** operation is used for choosing different functions depending on the value of the input to the **case**. Before illustrating the **case** we will introduce the success or fail datatype, otherwise known as **sf**.

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

Now we can use the **sf** datatype to return the value wrapped in **ss** in the root **node** of a tree or an error value if the tree is a **leaf**:

```
Charity>> def nodeValue : btree(A) -> sf(A) = tree =>
        { leaf => ff
        | node(num, _) => ss(num)
        }(tree).
Function added:  nodeValue : btree(A) -> sf(A)
Charity>> nodeValue(two11).
ss(succ(succ(zero))) : sf(nat)
Charity>> nodeValue(leaf).
ff : sf(A)
```

Note that the type signature of the function **nodeValue** was specified as having **btree(A)** as the domain and **sf(A)** as the codomain. Also note the use of an underscore, _, in the case function. The underscore is a variable that cannot be referenced later.

The **map** operation is used to perform an operation on the data inside a datatype. The **btree**s contain data of type **A** which can be operated on. For example, for a **btree(nat)** we could add n to every element:

```
Charity>> def addN = (tree, n) =>
               btree{num => addNat(num, n)}(tree).
Function added:  addN : btree(nat) * nat -> btree(nat)
Charity>> addN(two11, succ(zero)).
node(succ(succ(succ(zero))), (node(succ(succ(zero)), (leaf, leaf)),
node(succ(succ(zero)), (leaf, leaf)))) : btree(nat)
```

While **charity** does not have true higher order capability it is possible to pass functions into another function. The passed in functions are referred to as *macros*. They cannot lead to recursion. As an example we rewrite the **sumTree** function, which would only work with **btree(nat)**, so that it can sum a tree of any datatype (**btree(A)**).

```
Charity>> def gSumTree{add : A * A -> A} : btree(A) * A -> A  = (tree, seed) =>
                         {| leaf : () => seed
                          | node : (num, (leftT, rightT)) =>
                                        add(num, add(leftT, rightT))
                          |} (tree).
Function added:  gSumTree {A * A -> A} : btree(A) * A -> A
```

We see that the macro **add** is a binary operator (ie. `A * A -> A`), and the type of `gSumtree` resembles `sumTree` except it is defined on a general type `A` rather than the specific type `nat`. Note also that a `seed` value must be supplied for the `leaf` phrase.

Using `append` for `add` and `nil` (the empty list or `[]`) for `seed` we can evaluate a `btree` of lists. First, though, the `list` datatype:

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

Now the `append` function is defined:

```
Charity>> def append = (L1, L2) =>
                         {| nil : () => L2
                          | cons : (x, L) => cons(x, L)
                          |}(L1).
Function added:  append : list(A) * list(A) -> list(A)
```

Finally, we use gSumTree to "add" a `btree` of lists:

```
Charity>> gSumTree{append}(node([zero], (node([succ(zero)],(leaf,leaf)), leaf)), []).
[zero, succ(zero)] : list(nat)
```

Note the use of the list shorthand syntax. In this syntax, `[]` stands for `nil` and `[` $a_0, \ldots, a_n$`]` stands for `cons(`a_0 `, cons(` $\ldots, a_n$`nil))`. The syntax is always available in **charity** but it will not work properly until the `list` datatype is defined with the `nil` and `cons` constructors.

Whereas left datatypes can in a sense be associated with finite data (such as finite lists or the finite btree expressed above), right datatypes can in a sense be associated with infinite data. For example to define an infinite list:

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

which says that an `inflist(A)` has two destructors `head` and `tail` which when applied to an `inflist` return the head and tail respectively.

The associated operations with right datatypes are **unfold**, **record** and **map**. To define an infinite list (zero, succ(zero), succ(succ(zero)), ...), we use the **unfold** operation:

```
Charity>> def nats = () => (| x => head : x
                            |      tail : succ(x)
                            |)(zero).
Function added:  nats : 1 -> inflist(nat)
Charity>> head(nats).
zero : nat
Charity>> head(tail(nats)).
succ(zero) : nat
```

Since the display of an infinite amount of data is not feasible, the system will go into a `display-right-data mode`, which will incrementally display the right data:

```
Charity>> nats.
(head: ..., tail: ...)

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: ..., tail: ...))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: succ(zero), tail: (head: ..., tail: ...)))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: succ(zero), tail: (head: succ(succ(zero)), tail: (head: ..., tail: ...))))

Right display mode:
(q - quit, return - more) >>q
```

A **record** operation allows the addition of constants to the front of an infinite list. For example, to create an infinite list of nats prepended with two extra `zero`s:

```
Charity>> (head : zero, tail : (head : zero, tail : nats)).
(head: ..., tail: ...)

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: ..., tail: ...))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: ..., tail: ...)))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: zero, tail: (head: ..., tail: ...))))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: zero, tail: (head: succ(zero), tail: (head: ..., tail: ...)))))

Right display mode:
(q - quit, return - more) >>q
```

The **map** operation on right data has the same effect as the map operation on left data. For example, to generate the list of even positive numbers:

```
Charity>> inflist{x => addNat(x,x)}(nats).
(head: ..., tail: ...)

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: ..., tail: ...))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: succ(succ(zero)), tail: (head: ..., tail: ...)))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: succ(succ(zero)), tail: (head: succ(succ(succ(succ(zero)))),
tail: (head: ..., tail: ...))))

Right display mode:
(q - quit, return - more) >>q
```

Some syntactic concessions have been made for the benefit of the **charity** programmer. Parantheses that are not needed to resolve ambiguities are not generally required in an expression. For example:

```
Charity>> succ succ zero.
succ(succ(zero)) : nat
Charity>> sumTree leaf.
zero : nat
```

Datatype constructors or destructors that have the same type signature can be concatenated in their declaration:

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

Pattern matching can be used wherever one would expect to use a variable base. An inelegant way to prepend an `inflist` of `nat`s with two `zero`s:

```
Charity>> (| zero => head : zero
                  | tail : succ zero
           | succ zero => head : zero
                       | tail : succ succ zero
           | succ succ x => head : x
                         | tail : succ succ succ x
           |) zero.
(head: ..., tail: ...)

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: ..., tail: ...))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: ..., tail: ...)))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: zero, tail: (head: ..., tail: ...))))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: zero, tail: (head: succ(zero), tail: (head: ..., tail: ...)))))

Right display mode:
(q - quit, return - more) >>
(head: zero, tail: (head: zero, tail: (head: zero, tail: (head: succ(zero), tail: (head: succ(succ(zero)),
tail: (head: ..., tail: ...))))))

Right display mode:
(q - quit, return - more) >>q
```

Records can also be used as patterns.

```
Charity>> { (head : zero, tail : (head : zero, tail : _)) => false
          | _                                          => true
          } nats.
true : bool
Charity>> { (head : zero, tail : (head : zero, tail : _)) => false
          | _                                          => true
          } (head : zero, tail : nats).
false : bool
```

It is possible to use integers in **charity** but the correct datatypes must be entered first. The datatypes required are `digit, sign` and `int`.

```
Charity>> data digit -> C =
                    d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9 : 1 -> C.
Datatype added: digit
Charity>> data sign -> Y =
    positive | negative : 1 -> Y.
Datatype added: sign
Charity>> data int -> C =
    INT : sign * list(digit) -> C.
Datatype added: int
Charity>> 4096.
4096 : int
Charity>> -9345210538562446.
-9345210538562446 : int
```

It is also possible to use strings in **charity** but, again, the correct datatypes must be entered first. The datatypes required are `int`, defined above, along with `char` and `string` defined below:

```
Charity>> data char -> C =
    CHAR : list(digit) -> C.
Datatype added: char
Charity>> data string -> C =
    STRING : list(char) -> C.
Datatype added: string
Charity>> "Hello world."
"Hello world." : string
```

A character is defined by its ascii decimal representation. A string is a list of characters.

4 System Summary

The characters `Charity>>` are the prompt **charity** uses to indicate that it is ready for input. Input can contain embedded newlines. A `.` (period) tells **charity** to process the any input that is provided.

Only certain constructs can be entered at the system prompt. These constructs can be broken into five categories:

- datatypes
- function definitions
- functions
- commands
- queries

These five categories are discussed in following subsections. In addition, a subsection on patterns is included.

4.1 Datatypes

Charity uses the same general notion of a datatype that was introduced by Tatsuya Hagino in his thesis (University of Edinburgh). From the initial/left/inductive datatypes we can define the natural numbers, lists, binary trees, and other finite structures. From the final/right/coinductive datatypes we can define lazy tuples, infinite lists, infinite trees and other infinite structures.

Datatypes are defined using the keyword `data`. Both initial and final datatypes are defined using this keyword.

The general form of a `data` definition for initial datatypes is:

$$
\begin{array}{rrcl}
\texttt{data} & L(A_1, ..., A_m) & \texttt{->} & C \ \texttt{=} \\
& c_1 : E_1(A_1, ..., A_m, C) & \texttt{->} & C \\
| & c_2 : E_2(A_1, ..., A_m, C) & \texttt{->} & C \\
& \vdots & & \\
| & c_n : E_n(A_1, ..., A_m, C) & \texttt{->} & C.
\end{array}
$$

where E_i is one of: 1, C, A_i, $E_j * E_k$

Some examples of left datatypes are:

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

which says that the type bool has two constructors: `true` and `false` which do not have any arguments (ie. type `1`) and return values of type `bool`.

The following examples define the natural numbers, polymorphic lists, trees having nodes with a variable number of branches, and co-3ary-tuple.

```
Charity>> data nat -> C =
                 zero : 1 -> C
               | succ : C -> C.
Datatype added: nat
Charity>> data list(A) -> C =
                 nil : 1 -> C
               | cons: A * C -> C.
Datatype added: list
Charity>> data bush(A) -> C =
                 leaf : A -> C
               | node : list(C) -> C.
Datatype added: bush
Charity>> data multi(A,B,C) -> S =
                 one : A -> S
               | two : B -> S
               | thr : C -> S.
Datatype added: multi
```

The general form for a right datatype definition is:

$$
\begin{array}{rrcl}
\texttt{data} & D & \texttt{->} & R(B_1, ..., B_n) \ \texttt{=} \\
& d_1 : D & \texttt{->} & F_1(B_1, ..., B_n, D) \\
| & d_2 : D & \texttt{->} & F_2(B_1, ..., B_n, D) \\
& \vdots & & \\
| & d_m : D & \texttt{->} & F_m(B_1, ..., B_n, D).
\end{array}
$$

Some examples of right datatypes are:

```
Charity>> data D -> threeTuple(A,B,C) =
                 ex1 : D -> A
               | ex2 : D -> B
               | ex3 : D -> C.
Datatype added: threeTuple
```

which defines a 3-tuple, with projections `ex1`, `ex2` and `ex3`.

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

defines an infinite list, also known as a stream.

4.2 Patterns

Patterns are used in function definitions and in **charity**'s builtin constructs (the case, fold, map, record and unfold). Patterns are defined in this section and some examples are given in section ??.

The variable bases in the original version of **charity** are replaced by patterns in the current **charity** version. Patterns remove the need for the programmer to explicitly declare case statements to direct the result of a function; this logic is now implicit in the pattern.

Patterns must be complete. By this it is meant that any possible, well formed input must match at least one pattern. This removes a potential source of runtime errors thus preserving **charity**'s guarantee of (proper) termination.

4.2.1 Pattern definition

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
$$S \mapsto R(A) = d_i : S \mapsto F_i(A, R(A)) \qquad ,$$
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 $R(A)$.

4.2.2 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 $1 \leq i \leq n$.

4.2.3 Complete Sets of Patterns

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

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

4.3 Functions

4.3.1 Using The `def` Command

The **def** command allows a definition of a parameterized function to be made to the **charity** system.

The general form of the command is

$$\texttt{def}\ \mathit{ident}\ \{f_1 : A_1 \to B_1, ..., f_n : A_n \to B_n\}\ :\ A_0 \to B_0 = \begin{array}{ccc} p_1 & \mapsto & t_1 \\ & \vdots & \\ p_m & \mapsto & t_m \end{array}$$

where *ident* is the name equated to the definition, f_1 to f_n are maps to the function, p_1 to p_m are patterns as described in section ??, t_1 to t_m are functions as described in section ?? and $A_i \to B_i$ are typing constraints for the maps and function.

Type signatures are optional. If there are no macros then the braces ({ }) are optional as well.

If the type signature $A_0 \to B_0$ is present then $\{p_1, \ldots, p_n\} \lhd A_0$, and at least one pattern p_i must be of type A_0. The functions $t_1, \ldots, t_n$ must all be of type B_0. Otherwise $\{p_1, \ldots, p_n\}$ must be a complete set of patterns of some type P and the functions $t_1, \ldots, t_n$ must all be of some type Q.

For example,

```
Charity>> def double{f : A * A -> B} : A -> B = x => f(x,x).
Function added:  double {A * A -> C} : A -> C
```

defines a function, double, which takes a map (`f`) and a simple pattern `x` and returns the value of applying `f` to `(x,x)`.

In the following example, if the `threeTuple` record contains a value that is a `leaf`, `nil` or `zero` then the function `isEmpty` will return `true`, in all other cases it will return `false`.

```
Charity>> def isEmpty = (ex1 : leaf, ex2 : _   , ex3: _    ) => true
                     | (ex1 : _    , ex2 : nil, ex3: _    ) => true
                       | (ex1 : _    , ex2 : _   , ex3: zero) => true
                       | _                                  => false.
Function added:  isEmpty : threeTuple(A, B, C) -> bool
```

4.3.2 Definition of Functions

The following function definitions come from a technical report written by Cockett and Fukushima (?), with the exception of the case function, fold function, map function, and unfold function which have been redefined for pattern matching.

A **function** is defined by:

- () is a function of type type(()) = 1;

- if t is a function where $\mathrm{type}(t) = X \times Y$ then $\mathrm{p}_0(t)$ and $\mathrm{p}_1(t)$ are functions where $\mathrm{type}(\mathrm{p}_0(t)) = X$ and $\mathrm{type}(\mathrm{p}_1(t)) = Y$;
- if t_0 and t_1 are functions then (t_0, t_1) is a function where $\mathrm{type}((t_0, t_1)) = \mathrm{type}(t_0) \times \mathrm{type}(t_1)$;
- if t_1, ..., t_n are functions 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 $c_1(t_1)$, ..., $c_n(t_n)$ are functions where $\mathrm{type}(c_i(t_i)) = L(A)$;
- if t is a function where $\mathrm{type}(t) = P$, and $\{p_1, \ldots, p_m\} \triangleleft P$, and $t_1, \ldots, t_m$ are all functions 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 function (the case function) of type B; any variables in p_i are bound in t_i;
- if t is a function where $\mathrm{type}(t) = L(A)$, and c_i is a constructor of data type $L(A)$, and $\{p_{i1}, \ldots, p_{im_i}\} \triangleleft E_i(A, X)$, and $\forall ij \cdot t_{ij}$ is a function of type X then

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

  is a function (the fold) of type X; any variables in p_{ij} are bound in t_{ij};
- if t is a function where $\mathrm{type}(t) = L(A_1, \ldots, A_n)$, and $\{p_{i1}, \ldots, p_{im_i}\} \triangleleft A_i$, and $t_{i1}, \ldots, t_{im_i}$ are functions of type B_i then

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

  is a function (the map) of type $L(B_1, \ldots, B_n)$; any variables in p_{ij} are bound in t_{ij};
- if t is a function where $\mathrm{type}(t) = S$, and $\{p_1, \ldots, p_m\} \triangleleft S$, and $t_{i1}, \ldots, t_{in}$ are functions of type $F_i(A, S)$ then

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

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

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

$$\begin{pmatrix} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{pmatrix}$$

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

4.3.3 Pattern Matching Programs

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

???Does this cover off abstractions???

4.4 Command Summary

Commands can be broken into two subcategories. The first subcategory represents commands that tell **charity** to perform some action. The second subcategory of commands are used to set system parameters.

4.4.1 Action Commands

All action commands are prefixed with a : (colon). Entering `:?` will generate a help listing of action commands:

```
Charity>> :?.

Command Help:
   [r | R | rf | readfile] <fname>    - reads <fname> into charity
   [q | Q | quit]                     - exits the charity interpreter
   ?                                  - produces this help listing
```

There are two action commands, one to read in input files and one to exit **charity**.

The command to read input files into **charity** has four (count them!) different forms.

`[r | R | rf | readfile] <fname>`

The argument `<fname>` is required and must be enclosed in double quotes (").

The exit command has only three forms:

`[q | Q | quit]`

4.4.2 Set System Parameter Commands

All set system parameter commands are prefixed with `:set`. Entering `:set?` at the **charity** prompt will generate a help listing of set commands:

```
Charity>> :set?.

Set Command Help :
   replace functions [true | false] - if true don't display replacement warnings
   searchpath "<dirname>", ...   - sets up search path for charity files
   appendpath "<dirname>", ...   - appends <dirname>(s) to search path
   ?                               - produces this help listing
```

Giving the `replace functions true` command allows functions to be replaced silently when they are redefined. The default action in **charity** is to prompt the user before redefining functions, i.e. `replace functions false`. Redefining functions may cause **charity** to act unpredictably and is not recommended.

Charity maintains a list of paths where a path is a directory name. All directories in the path list are searched for an input file given as argument to a `readfile` command. The directories are searched in the order they appear in the path list. The default path list in **charity** contains only the current directory. To define a new path list use the `searchpath` command. To append directories to the path list use the `appendpath` command. For both commands be sure to put `''` around each directory name and separate the directories with commas.

4.5 Queries

Queries are used to get information about the state of the **charity** system. All queries are prefixed with `?` (question mark). Entering `??` at the **charity** prompt will generate a listing of query commands:

```
Charity>> ??.

Query Command Help :
   <ident>          - shows type info for <ident>
   comb <ident>     - shows type info for <ident>. In addition,
                      shows combinator code for functions and
                      shows operation combinator types for datatypes.
   dump table       - shows all entries in symbol table
   mem use          - gives a listing of memory usage
   set replace      - shows status of replacement prompts
   set searchpath   - shows search path list
   ?                - produces this help listing
```

4.6 Scoping Rules

`This section is not complete and may not be accurate.`

Scoping rules are used to determine the meaning of identifiers. An identifier can be a *variable, macro name* or *function.*

An identifier appearing in a pattern is first checked to see if it is a constructor or destructor. If it is not then it is taken to be a variable. This can be the source of a subtle error. The following definition of `foo` can mean two different things depending on whether datatype `bool` has been defined or not (assume datatype `nat` has been defined).

```
Charity>> def foo = true  => zero
                  | false => succ(zero).
WARNING: Redundant patterns.
Function added:  foo : A -> nat
Charity>>  data bool -> C = true | false : 1 -> C.
Datatype added: bool
Charity>> foo(false).
zero : nat
```

If datatype `bool` had been defined first:

```
Charity>>  data bool -> C = true | false : 1 -> C.
Datatype added: bool
Charity>>  def foo = true => zero | false => succ(zero).
Function added:  foo : bool -> nat
Charity>>  foo(false).
succ(zero) : nat
```

An identifier appearing outside a pattern is attempted to be interpreted first as a macro, then as a function and, finally, as a variable if the context allows it.

Thus function and macro names can be redefined in a pattern. Given a function `foo`, the following code redefines `foo` as a variable:

```

```

4.7 ()

The () represents the null argument and has type `1`.

4.8 Shorthands

Some alternative notation for special cases are provided in order to improve readability of programs.

4.8.1 Lists

A shorthand is provided for constructing lists. Eg. $[\,], [zero], [b0(zero), b1(false)], ...$

$$[e_1, e_2, ..., e_n] \equiv cons(e_1, cons(e_2, ..., cons(e_n, nil)...))$$

$$[\,] = nil$$

4.8.2 Int

Eg. $0, 1, 24124,\ 2343, ...$

$$0 \equiv INT(b1, [d0])$$

$$123 \equiv INT(b0, [d1, d2, d3])$$

4.8.3 Strings

Eg. "hello", "", ...

$$\text{""} \equiv STRING([])$$

$$\text{"}hi\text{"} \equiv STRING([CHAR([d1, d0, d4]), CHAR([d1, d0, d5])])$$

4.8.4 Underscores

Underscores (ie. _) can be used in patterns just as variable would be. However, unlike a variable, their values cannot be referenced later. For example, if we wanted to test to see if a list was empty of not then we do not care about the values in the list:

```
> def is_nil (L) =
+    { nil  => true
+    | cons(_) => false
+    } (L).
```

A Charity Grammar

This appendix contains a description of the extended BNF grammar for **charity** version β1.00.

A.1 Terminals

The set of terminals for **charity** is the union of the following sets:

- {-> , = , '(' , ')' , '|' , , , : , '{' , '}' , _ , * , + , '[' , ']' , '(|' , '|)' , '{|' , '|}' , => , " , . }
- {**data** , **def** , **:set** , **?** , **quit** , **readfile** , **rf** , **replace** , **searchpath** , **appendpath** , **functions** , **datatypes** , **structors** , **all** , **true** , **false** , **comb** , **dump table** , **show mem** }
- set of printable characters including all the uppercase and lowercase letters in the English alphabet
- set of digits

A.2 Miscellaneous Nonterminals

Empty is the empty string.

String ::= nonterminal representing a concatenation of one or more elements of the printable character set.

Identifier ::= Alpha { Alpha | Digit | _ | ' | **$** }

Digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**

Alpha ::= nonterminal representing a lowercase or uppercase letter of the English alphabet.

IntNeg ::= - IntPos

IntPos ::= Digit {Digit}

Start ::=
    Data.[a]
  | Def.[b]
  | Term.[c]
  | Command.[d]

[a]Data is defined in section ??
[b]Def is defined in section ??
[c]Term is defined in section ??
[d]Command is defined in section ??

A.3 Data Definition Nonterminals

Data ::= **data** Identifier TypeVars -> Identifier TypeVars = StructList

TypeVars ::=
    '(' IdList ')'
  | '(' ')'
  | Empty

StructList ::=
    Struct '|' StructList
  | Struct

Struct ::=
    Identifier '|' Struct
  | Identifier **:** TypeSig[a]

IdList ::=
    Identifier **,** IdList
  | Identifier

[a]TypeSig is defined in section ??

A.4 Function Definition Nonterminals

Def ::=
  def Identifier Macros = Cases[a]
  | **def** Identifier Macros **:** TypeSig = Cases[b]
  | **def** Identifier Macros VarBase = Term[c]

[a] Cases are defined in section ??
[b] Cases are defined in section ??
[c] This function definition syntax is obsolote. It is maintained for this version only to provide backwards compatibility.

Macros ::=
  '{' MacroList '}'
  | '{' '}'
  | Empty

MacroList ::=
  Macro , MacroList
  | Macro

Macro ::=
  Identifier **:** TypeSig
  | Identifier

VarBase[a] ::=
  '(' VarBasex , VarBasex ')'
  | '(' VarBasex ')'
  | '(' ')'

VarBasex[b] ::=
  '(' VarBasex , VarBasex ')'
  | '(' VarBasex ')'
  | '(' ')'
  | Identifier
  | _

[a] VarBase is part of an obsolete function definition
[b] VarBasex is part of an obsolete function definition

A.5 Type Signature Nonterminals

TypeSig ::= Dom -> Dom

Dom ::=
  '(' Dom ')'
  | Identifier '(' DomList ')'
  | Identifier
  | Dom * Dom
  | Dom + Dom
  | **1**

DomList ::=
  Dom , DomList
  | Dom

A.6 Term Nonterminals

```
Term ::=
        Termx
      | Function Term
      | TermExt

Termx ::=
        '(' Term , Term ')'
      | '(' Term ')'
      | '(' Records ')'
      | '(' ')'

TermExt ::=
        '[' TermList ']'
      | '[' ']'
      | Identifier
      | " String "
      | IntNeg
      | IntPos

TermList ::=
        Term
      | Term , TermList

Function ::=
        '{' Cases '}'
      | '(|' Fold '|)'
      | '{|' Unfold '|}'
      | Identifier '{' FunctionMacros '}'
      | Identifier

FunctionMacros ::=
        Function
      | Cases
      | Function , FunctionMacros
      | Cases , FunctionMacros
```

A.6.1 Case Nonterminals

```
Cases ::=
        CasePhrase
      | CasePhrase '|' Cases
```

CasePhrase ::= Patt[a] => Term

[a] Patt is defined in section ??

A.6.2 Fold Nonterminals

```
Fold ::=
        Identifier : CasePhrase
      | Identifier : CasePhrase '|' FoldPhrase

FoldPhrase ::=
        Fold
      | CasePhrase
      | CasePhrase '|' FoldPhrase
```

A.6.3 Record Nonterminals

Records ::=
  Identifier **:** Term
  | Identifier **:** Term , Records

A.6.4 Unfold Nonterminals

Unfold ::=
  UnfoldSentence
  | Fold

UnfoldSentence ::=
  Patt[a] **=>** Identifier **:** Term
  | Patt[b] **=>** Identifier **:** Term '|' UnfoldPhrase

UnfoldPhrase ::=
  UnfoldSentence
  | Identifier **:** Term
  | Identifier **:** Term '|' UnfoldPhrase

[a]Patt is defined in section ??
[b]Patt is defined in section ??

A.7 Pattern Nonterminals

Patt ::=
  Pattx
  | Identifier Patt
  | PattExt

Pattx ::=
  '(' ')'
  | '(' Patt ')'
  | '(' Patt , Patt ')'
  | '(' PattDestr ')'

PattExt ::=
  '{' PattList '}'
  | '{' '}'
  | Identifier
  | **_**
  | " String "
  | IntNeg
  | IntPos

PattDestr ::=
  Identifier **:** Patt
  | Identifier **:** Patt , PattDestr

PattList ::=
  Patt
  | Patt , PattList

A.8 Command Nonterminals

Command ::=
  : MetaCommand
  | **:set** SetCommand
  | **?** Query

A.8.1 Meta Command Nonterminals

MetaCommand ::=
  ReadFile String
  | **quit**
  | **?**

ReadFile ::=
  readfile
  | **rf**
  | **R**
  | **r**

A.8.2 State Command Nonterminals

SetCommand ::=
    replace SymtabEntry Bool
  | **searchpath** DirList
  | **appendpath** DirList
  | **?**

SymtabEntry ::=
    functions
  | **datatypes**
  | **structors**
  | **all**

Bool ::=
    true
  | **false**

DirList ::=
    String
  | String , DirList

A.8.3 Query Command Nonterminals

Query ::=
    QueryString
  | **set replace**
  | **set searchpath**
  | **comb** QueryString
  | **dump table**
  | **show mem**
  | **?**

QueryString ::=
    Identifier
  | *
  | +
  | 1