DRAFT: Charity User Manual

Tom Fukushima
Department of Computer Science
University of Calgary
Calgary, Alberta T2N 1N4
CANADA
email: fukushim@cpsc.ucalgary.ca

November 24, 1991

1 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 2. Section 3 describes our notational conventions for types and combinator expressions. Section 4 describes the available commands. Section 5 describes the `data` command in more detail. Section 6 describes the syntax of the term-logic in more detail.

Programs can be written at two levels: the term-logic and the categorical combinator levels. The term-logic level is the level which is useful for actual programming. The categorical combinators are the "machine code" for the abstract machine on which evaluation takes place. This document will concentrate on the use of the term-logic.

Changes in the syntax of the language that have occured since the last release are summarized in appendix A.

In order to give the user some basic operations, some predefined datatypes and combinators are provided when the system is started. These are described in appendix B. If the user does not want these definitions then the `restart_base` command will eliminate them.

The charity system is available via anonymous ftp from `cpsc.ucalgary.ca` in the pub/charity directory. Please send any inquiries, comments and bugs to the author.

2 Example session

We give a taste of the charity system through the use of an example in which operations on a binary tree (btree) are performed. After charity is started it will initially display:

```
Charity ready (V0.3)
>
```

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

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

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

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). The `+` (plus) signals that a command is being continued over several lines.

We can now create some `btrees`:

```
> leaf.
leaf : btree('a)
3 machine operations,  0.0 cpu sec
> node(1,(leaf,leaf)).
node(1,(leaf,leaf)) : btree(int)
26 machine operations,  0.0 cpu sec
```

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 gives it a most general type `'a`. When the command is an expression, the system executes the expression and displayes the result and execution statistics. The second `btree` is a node with an integer value and whose subtrees are `leaf`s.

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

```
> def one = node(1,(leaf,leaf)).
> one.
node(1,(leaf,leaf)) : btree(int)
28 machine operations,  0.0 cpu sec
```

Subsequent uses of the combinator `one` will be equivalent to using `node(1,(leaf,leaf))`. To create a tree with a root `node` containing `2` and `one` as the subtrees we could type:

```
> def two11 = node(2,(one,one)).
> two11.
node(2,(node(1,(leaf,leaf)),node(1,(leaf,leaf)))) : btree(int)
78 machine operations,  0.0 cpu sec
```

Supplied with every datatype declaration are three operations. With the left datatypes (typename is to the left of the arrow in the `data` declaration) we get a **fold**, **case** and **map** operation for free. A **fold** operation replaces the constructors of the datatype by functions and evaluates the resulting expression. For example, to add up all of the values in the nodes of a tree:

```
> def sumtree(T) =
+     {| leaf : () => 0
+      | node : (n,(sumL,sumR)) => add_int(n,(add_int(sumL,sumR)))
+      |} (T).
> ?sumtree.
sumtree : btree(int) -> int
> sumtree(leaf).
0 : int
22 machine operations,  0.0 cpu sec
> sumtree(two11).
4 : int
1769 machine operations,  0.680000 cpu sec
```

The `def` command says that `sumtree` takes one value (`T`). The `T` is applied to a fold operation for `btree` which says: recurse over the `btree` returning `0` 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:

```
sumtree(T) =
    case T of
        leaf () => 0
      | node (n,(tL,tR)) => add_int(n,add_int(sumtree(tL),sumtree(tR)))
```

The query command (`?`) is used to get type information on a combinator. (Using `set verbose on` will cause this information to be displayed automatically). Thus `sumtree` takes a `btree(int)` and returns a result of type `int`.

To give an example of a combinator which takes an abstraction as an argument, we could create a generic `sumtree`:

```
> set verbose on.
Verbose true
> def Gsumtree{add}(T,zero) =
+     {| leaf : () => zero
+      | node : (n,(sumL,sumR)) => add(n,(add(sumL,sumR)))
+      |} (T).
Added: Gsumtree {('a * 'a) * 'b -> 'a} : (btree('a) * 'a) * 'b -> 'a
```

Thus ignoring the `'b` (ie. strength) [1] in the typing, we see that the abstraction `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 `int`. Note also that a `zero` value must be supplied for the `leaf` phrase.

Using `append` for `add` and `[]` (the empty list) for `zero` we can evaluate:

```
> Gsumtree{x => append(x)}(node(  ["h"]
+                               ,  ( node(["i"],(leaf,leaf))
+                                  , leaf
+                                  )
+                               )
+                          ,[]).
["h","i"] : list(string)
274 machine operations,  0.020000 cpu sec
```

Note that the append has to be made explicitly into an abstraction (ie. `x => append(x)`).

The **case** operation is efficient for doing a single destruct operation on some data. For example to obtain the value in the root `node` of a tree or an error value if the tree is a `leaf`:

```
> def rootlook(T) =
+     { leaf => b0
+     | node(n,_) => b1(n)
+     } (T).
Added: rootlook : btree('a) -> 1 + 'a
> rootlook(two11).
b1(2) : 1 + int
87 machine operations,  0.0 cpu sec
> rootlook(leaf).
b0 : 1 + 'a
10 machine operations,  0.0 cpu sec
```

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(int)` we could add `n` to every element:

```
> def add_n(T,n) = btree{x => add_int(x,n)}(T).
Added: add_n : btree(int) * int -> btree(int)
> add_n(two11,3).
node(5,(node(4,(leaf,leaf)),node(4,(leaf,leaf)))) : btree(int)
940 machine operations,  0.350000 cpu sec
```

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:

[1] Strength allows the combinator to reference the environment of any combinator that "calls" it. For system defined combinators that require strength, the type `'$` is used

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

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

The associated operations with right datatypes are **unfold**, **record** and **map**. To define an infinite list (0, 1, 2, ...), we use the **unfold** operation:

```
> def nats = (| x =>
+               head : x
+             | tail : succ_int(x)
+             |) (0).
Added: nats : 'a -> inflist(int)
> head(nats).
0 : int
24 machine operations,  0.0 cpu sec
> head(tail(nats)).
1 : int
131 machine operations,  0.060000 cpu sec
```

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:

```
> nats.
... : inflist(int)
21 machine operations,  0.0 cpu sec
Entering display-right-data mode...

(head:0,tail:...)
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:1,tail:...))
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:1,tail:(head:2,tail:...)))
Press <Return> to display more or 'q' to quit: 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 `0`s:

```
>(head: 0, tail: (head : 0, tail : nats)).
... : inflist(int)
6 machine operations,  0.0 cpu sec
Entering display-right-data mode...

(head:0,tail:...)
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:0,tail:...))
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:0,tail:(head:0,tail:...)))
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:0,tail:(head:0,tail:(head:1,tail:...))))
Press <Return> to display more or 'q' to quit: 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:

```
> inflist{x => add_int(x,x)}(nats).
... : inflist(int)
25 machine operations,  0.0 cpu sec
Entering display-right-data mode...

(head:0,tail:...)
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:2,tail:...))
Press <Return> to display more or 'q' to quit:

(head:0,tail:(head:2,tail:(head:4,tail:...)))
Press <Return> to display more or 'q' to quit: q
```

To terminate the charity session type <ctrl>-d.

3 Notation

This section describes the notational conventions that are used for type definitions, and categorical combinator expressions.

3.1 Types

All expressions must be well typed. The type of a combinator (ie. program) can be queried using the `?` command described below.

The form of a type is

$$ident\{p_1, ..., p_n\} : t$$

where *ident* is the name of the combinator, p_i is the type of the ith abstraction to *ident* and t is the resulting type.

For example,

```
def mydef {f} (a) = f(a,a).
```

would define a function of type

$$\{(\alpha \times \alpha) \times \beta \to \gamma\} : \alpha \times \beta \to \gamma$$

which says that (ignoring strength) `mydef` which is of type $\alpha \to \gamma$ takes, as an argument, a map `f` which has type $\alpha \times \alpha \to \gamma$.

Strength is used to allow combinators to have access to the environment in which the combinator is used. When a combinator does not take any abstractions as parameters, strength is not required. For example, for the definition

```
def mydef(a) = (a,a).
```

`mydef` will have type $\alpha \to \alpha * \alpha$.

Characters not available on the keyboard are represented using the following methods. The greek characters (ie. $\alpha, \beta, \gamma ...$) for variable types are represented by prefixing an identifier with a "`'`" (quote). The "$\to$" is replaced by a "`->`" (dash, greater than) and the "$\times$" by "`*`" (asterisk).

3.2 Categorical combinator expressions

This section can be skipped if the user is only using the term-logic. A combinator expression is similar to an expression in category theory with ";" as the composition symbol.

Composition is read from left to right so if $f : \alpha \to \beta$ and $g : \beta \to \gamma$ then $f; g : \alpha \to \gamma$ (ie. f then g).

The predefined combinators (after a `restart_base` command) and their corresponding definitions are

Combinator	Category Theory	Description
`id`	1, id	identity
`!`	!	terminal map
`<_,_>`	$< \_, \_ >$	cartesian product
`p0`	Π	first projection
`p1`	Π'	second projection
`map_1`	Θ^1	$\Theta^1 : 1 \times \alpha \to 1$
`map_id`	Θ^{id}	$\Theta^{id} : \alpha \times \beta \to \alpha \times \beta$
`map_prod`	$\Theta^\times; f \times g$	$\Theta^\times : (\alpha \times \beta) \times \gamma \to (\alpha \times \gamma) \times (\beta \times \gamma)$

Combinators with parameters are written $comb\{p_1, p_2, ..., p_n\}$ where p_i are the parameters.
Given the definitions:

```
> data coprod(A,B) -> C =
+     b0 : A -> C
+   | b1 : B -> C.
> data list(A) -> C =
+     nil : 1 -> C
+   | cons : A * C -> C.
```

A shorthand for coproduct is supplied:

$$<x|y> \equiv \text{case\_coprod}\{x, y\}.$$

A shorthand for lists is supplied:

$$[a, b, ..., c] \equiv <a, <b, ... <c, nil >; cons...>; cons >; cons$$

4 Command Summary

The `>` (greater than sign) is the system prompt that signals that **charity** is ready for a command. Commands can be continued over lines in which case a `+` (plus sign) prompt will be given. All commands are terminated with a `.` (period).

Summary of the available commands are:

Command name	Brief description	Section
`?`	query information on a combinator or definition	4.1
`cdef`	equate a name with a combinator expression	4.2
`ceval`	evaluate a combinator expression	4.3
`data`	define a datatype to the system	5
`def`	equate a name with a program	6.1
`dump_cdef_table`	show all defined names	4.4
`dump_table`	show all defined names and combinators	4.5
expression	evaluate a program	6.2
`restart`	erase user definitions and datatypes	4.6
`restart_base`	start system with no predefined definitions	4.7
`set trace`	set execution trace on (off)	4.9
`set verbose`	(do not) display informational messages	4.10
`set safety`	(dis)allow user redefinitions	4.8
`readfile`	read a prepared input file	4.11

4.1 Query

In order for expressions to be accepted for rewriting it must be typeable. Using `?`, the signature of combinators and definitions can be queried.

For example, to see the type of add:

```
> ? add.
add : nat * nat -> nat
```

If verbose mode is on then the combinator code is also shown.

4.2 cdef

The `cdef` (**c**ategorical combinator **def**inition) command allows complex combinator expressions to be equated to some name. Definitions can be parameterized so that some variables may occur in the expression.

The general format is

$$cdef\ id\{parm_1, ..., parm_n\} = combinator\ expression.$$

where the combinator expression can only use variables in the parameter list, already existing combinators or definitions.

The braces are not required if the parameter list is empty.

For example to define reverse_pair which swaps the components of a pair.

```
> cdef reverse_pair {pair} = pair ; <p1,p0>.
Added def: reverse_pair  {'a -> 'b * 'c} : 'a -> 'c * 'b
```

4.3 ceval

The `ceval` command typechecks a combinator expression and rewrites it.

For example, to swap the elements of a pair of lists:

```
> ceval reverse_pair{<[],[!]>}.

typing : 1 -> list(1) * list('a)

<[!] , []>
```

4.4 dump_cdef_table

To see the combinator definitions in the system:

```
> dump_cdef_table.
...
```

Note that the parameters have been translated into positional markers.

4.5 dump_table

To see the signatures and types of all of the combinators and definitions on the system:

```
> dump_table.
...
```

4.6 restart

`restart` restarts the system by reinitializing all of the tables to the basic set of combinators and predefined definitions and datatypes. All user combinators and definitions will be lost.

Usage:

```
> restart.
```

4.7 restart_base

`restart_base` restarts the system by reinitializing all of the tables. No predefined datatypes (ie. nat, bool, list) will be defined, nor will the predefined definitions (ie. add, mul, sub, or, ...) be supplied. All user combinators and definitions will be lost.

Usage:

```
> restart_base.
```

4.8 safety

The user may periodically want to redefine a system definition without having to restart the system. The `safety` command determines whether the user can redefine these or not. Initially, on system start-up the safety is on, so no combinator definitions can be redefined. Set safety off to allow redefinitions. When a combinator is redefined a warning will be given and there is no guarantee that all programs will execute correctly.

Usage:

```
> set safety on.
Safety true, existing definitions not changeable.
> set safety off.
Safety false, definitions are changeable.
```

4.9 trace

Enables/disables the ability to see the stacks of the abstract machine as an expression is rewritten. NOT recommended to be on for the evaluation of complex expressions.

Usage:

```
> set trace on.
...
> set trace off.
...
```

4.10 Verbose

If verbose is on the information about the combinators and definitions going into the system will be displayed.

Usage:

```
> set verbose on.
...
> set verbose off.
...
```

4.11 readfile

The `readfile` command allows the processing of a prepared input file which contains charity commands.

Usage:

```
> readfile "filename".
...
file read
closing filename
```

5 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 datatypes we can define the natural numbers, lists, binary trees, and other finite structures. From the final/right datatypes we can define lazy tuples, infinite lists, infinite trees and other infinite structures.

Both initial and final datatypes are definable through this command.

Some examples of left datatypes are:

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

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

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


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


> data bush(a) -> C =
+      leaf : a -> C
+    | node : list(bush) -> C.


> data multi(a,b,c) -> S =
+      one : a -> S
+    | two : b -> S
+    | thr : c -> S.
```

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

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

Some examples of right datatypes are:

```
> data D -> three_tuple(a,b,c) =
+     ex1 : D -> a
+   | ex2 : D -> b
+   | ex3 : D -> c.
```

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

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

defines an infinite list.

The general form for a right datatype definition is:

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

6 Term-logic programs

Programming at the categorical combinator level is quite hard so the term-logic was developed to facilitate program development. Programs written using the term-logic can be defined using the `def` command and evaluated by entering a closed expression as a command.

6.1 Def

The `def` command allows a definition of a parameterized map/function to be made to the system. The general form of the command is

$$def\ ident\{f_1, ..., f_n\}(var\_base) = expr$$

where ident is the name equated to the definition, f_1 to f_n are maps to the term, the *var_base* is a tuple[2], and *expr* is formed from the operations described below. If there are no f_is then the braces are optional. If the *var_base* is empty then the parenthesis are optional.

For example,

```
> def double {f} (a) = f(a,a).
Added: double {('a * 'a) * 'b -> 'c} : 'a * 'b -> 'c
```

defines double which takes a map (`f`) and an argument (`a`) and returns the value of applying `f` to `(a,a)`.

6.2 Execute

Typing a closed expression in as a command causes the evaluation of that expression. For example, using the previous definition, we could double the number 1 by:

```
> double{x => add_int(x)} (1).
2 : int
291 machine operations,  0.130000 cpu sec
```

6.3 Term-logic expressions

The possible types of term-logic operations that may be performed are: fold, unfold, case, record, map, abstraction, constructor/destructor application and combinator call (macro substitution).

6.3.1 Fold

The fold is an operation on left data

$$
\left\{
\begin{array}{rcl}
c_1 : patt_1 & \mapsto & expr_1 \\
& \vdots & \\
c_m : patt_m & \mapsto & expr_m
\end{array}
\right\} (expr)
$$

which in our notation becomes

[2]Currently, only tuples formed from pairs are allowed

$$
\begin{array}{lcll}
\{| & c_1 : patt_1 & \texttt{=>} & expr_1 \\
| & c_2 : patt_2 & \texttt{=>} & expr_2 \\
 & & \vdots & \\
| & c_m : patt_m & \texttt{=>} & expr_m \\
|\} & (expr) & &
\end{array}
$$

where $(patt_i\texttt{=>}expr_i) : E_i(A_1, ..., A_n, C) \to C$; $expr : L(A_1, ..., A_n)$; and the type of the statement is C.

6.3.2 Unfold

The unfold is an operation on right data. Only one variable base (ie. *patt*) Is used for the whole expression since the destructors, d_i, all have the same variable base.

$$
\left(\left| patt \mapsto \begin{array}{c} d_1 : expr_1 \\ \vdots \\ d_m : expr_m \end{array} \right| \right) (expr)
$$

which in our notation becomes

$$
\begin{array}{lcll}
\texttt{(|} & patt & \texttt{=>} & \\
 & d_1 : & & expr_1 \\
| & d_2 : & & expr_2 \\
 & \vdots & & \\
| & d_m : & & expr_m \\
\texttt{|)} & (expr) & &
\end{array}
$$

where $(patt\texttt{=>}expr_i) : D \to F_i(B_1, ..., B_n, D)$; $expr : D$; and the type of the statement is $R(B_1, ..., B_n)$.

6.3.3 Case

The case statement

$$
\left\{ \begin{array}{lcl} c_1\texttt{(}patt_1\texttt{)} & \mapsto & expr_1 \\ & \vdots & \\ c_m\texttt{(}patt_m\texttt{)} & \mapsto & expr_m \end{array} \right\} (expr)
$$

which in our notation becomes

$$
\begin{array}{lcll}
\{ & c_1(patt_1) & \texttt{=>} & expr_1 \\
| & c_2(patt_2) & \texttt{=>} & expr_2 \\
 & & \vdots & \\
| & c_m(patt_m) & \texttt{=>} & expr_m \\
\} & (expr) & &
\end{array}
$$

where $(c_i(patt_i)\texttt{=>}expr_i) : E_i(A_1, ..., A_n, L(A_1, ..., A_n)) \to T$; $expr : L(A_1, ..., A_n)$ and the type of the statement is T.

6.3.4 Record

A lazy labeled tuple

$$\left(\begin{array}{c} d_1 : expr_1 \\ \vdots \\ d_m : expr_m \end{array} \right)$$

which in our notation becomes

$$\begin{array}{lcl} \texttt{(} & d_1 & : \quad expr_1 \\ \texttt{,} & d_2 & : \quad expr_2 \\ & \vdots & \\ \texttt{,} & d_m & : \quad expr_m \\ \texttt{)} & & \end{array}$$

where $(expr_i) : F_i(B_1, ..., B_n, L(B_1, ..., B_n))$ and the type of the statement is $R(B_1, ..., B_n)$.

Note that products are an eager record with no labels. That is `(`x`,`y`)` is the same as $eager($`(p0:`x`, p1:`y`)`$)$, where *eager* causes expressions x and y to evaluate.

6.3.5 Map

The map operation can be used on either left or right data. For example, for a type $F(\alpha_1, ..., \alpha_n)$:

$$F \left\{ \begin{array}{lcl} patt_1 & \mapsto & expr_1 \\ & \vdots & \\ patt_n & \mapsto & expr_n \end{array} \right\} (expr)$$

which in our notation becomes

$$\begin{array}{rlcl} F\texttt{\{} & patt_1 & \texttt{=>} & expr_1 \\ \texttt{,} & patt_2 & \texttt{=>} & expr_2 \\ & \vdots & & \\ \texttt{,} & patt_n & \texttt{=>} & expr_n \\ \texttt{\}} & (expr). & & \end{array}$$

where $(patt_i\texttt{=>}expr_i) : \alpha_i \to \beta_i$ $expr : F(\alpha_1, ..., \alpha_n)$; and the type of the statement is $F(\beta_1, ..., \beta_n)$.

6.3.6 Abstraction

Abstractions are useful for breaking up pairs. For example if `t` had type $\alpha \times (\beta \times \gamma)$ then the statement

$$\texttt{\{(x,y) =>} \text{ expr } \texttt{\} (t)}$$

would allow expr to use variables `x` of type α and `y` of type $\beta \times \gamma$.

6.3.7 Constructor/destructor application

When constructors are applied to values of the proper type they can create data values. For example, `nil` has type $1 \to list(\alpha)$, so when applied to a null argument (ie. `()`) returns a list. Similarily, `head` has type $inflist(\alpha) \to \alpha$, and when applied to an infinite list of type α returns a value of type α.

6.3.8 Combinator call

When a user defined combinator is used in a term-logic expression, the equated term-logic expression is substituted with the proper substitutions for variables.

6.4 Variable binding

Variables in a definition are bound to the closest enclosing static definition of it For example

```
> def t {f} (f) =
+     {f => f} (f).
```

The first `f` (in the braces) is not accessible in the body of the definition. The last `f` is bound to the second `f`. The fourth `f` is bound to the third `f`.

6.5 ()

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

6.6 Shorthands

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

6.6.1 Lists

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

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

$$[\,] = nil$$

6.6.2 Int

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

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

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

6.6.3 Strings

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

$$"" \equiv STRING([])$$

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

6.6.4 if ... then ... else

Eg. if eq(x,y) then "equals" else "not equals".

$$\text{if } expr \text{ then } expr_T \text{ else } expr_F \equiv \left\{ \begin{array}{ccc} \text{true} & \mapsto & expr_T \\ \text{false} & \mapsto & expr_F \end{array} \right\} (expr)$$

6.6.5 Underscores

Underscores (ie. _) can be used in the variable bases to ignore values. For example, if we wanted to test to see if a list was empty of not, we do not care about the values in the list:

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

Underscores can be used in case and fold statements to provide a default action for the unspecified constructors. Using the same example as above:

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

7 Examples

Check the *.ch files in the EXAMPLES directory which currently contains:

Filename	Description
BINARY.ch	binary number and some operations on them
INT.ch	integers and operations on them (builtin)
LIST.ch	some list operations
STRING.ch	strings and operations on them (builtin)
UTILS.ch	basic utilities (builtin)
ackermann.ch	a function which is not primitive recursive
bubsort.ch	bubble sort
bubsort.tst.ch	test of the bubble sort
comb.eg.ch	example of coding using the categorical combinators
evenodd.ch	calculate whether a number is even or odd
factorial.ch	calculate n!
fibonacci.ch	calculate the fibonacci sequence
group.ch	group the elements in a list
group.tst.ch	test of group.ch
inflist.ch	define an infinite list
lexer.ch	define a simple lexer
onetwo.ch	datatype test
pascal.ch	pascals triangle
primes.ch	calculate prime numbers using int
primes.nat.ch	calculate prime numbers using nat
qs.ch	quicksort
qs.tst.ch	test the quicksort
rand.ch	random number generator
sort.ch	an efficient sort
sort.tst.ch	test of sort
sublists.ch	calculate all sublists of a list
theorems_for_free.ch	examples from Philip Wadler's paper
treepaths.ch	
unify1.ch	unification algorithm
zip_trees.ch	as opposed to zipping two lists together
zip_trees.tst.ch	test of zip_trees.

A Changes from the previous release

Here we summarize the changes from the previous release.

- The use of quotes (ie. `"` and `‘`) have been eliminated for all commands except for readfile.
- The `use` command has been replaced by the `readfile` command.
- The case factorizer in the term-logic must have at least one set of parenthesis following the label. For example, a case which could have been written `nil =>` expr previously must now be written `nil () =>` expr. This is to emphasize the fact that `nil` is a label and that there must be a variable base (in this case null).
- The `fold` and `map` operations are new.
- `?` command replaces the `ctype` command.
- `cdef` replaces `macro`.
- `ceval` replaces `eval`.
- `evalt` is no longer needed.
- `def` replaces `term`.

NOTE: The program `convert`, which is also available via anonymous ftp from cpsc.ucalgary.ca, will automatically convert programs in the old syntax to the new syntax.

B Predefined datatypes and definitions

The list of predefined datatypes and definitions:

B.1 Bools, lists and natural numbers

See the file EXAMPLES/UTILS.ch.

```
(* constructors *)
nil  : 1 -> list('a)
cons  : 'a * list('a) -> list('a)
true  : 1 -> bool
false  : 1 -> bool
zero  : 1 -> nat
succ  : nat -> nat

(* logical operations *)
or  : bool * bool -> bool
not  : bool -> bool
and  : bool * bool -> bool
imp  : bool * bool -> bool
eqv  : bool * bool -> bool
xor  : bool * bool -> bool

(* arithmetic operations *)
add  : nat * nat -> nat
pred  : nat -> nat
sub  : nat * nat -> nat
mul  : nat * nat -> nat
max  : nat * nat -> nat
min  : nat * nat -> nat
div_mod  : nat * nat -> 1 + (nat * nat)
div  : nat * nat -> 1 + nat
mod  : nat * nat -> 1 + nat
exp  : nat * nat -> nat

(* comparison operations *)
ge  : nat * nat -> bool
le  : nat * nat -> bool
eq  : nat * nat -> bool
gt  : nat * nat -> bool
lt  : nat * nat -> bool

(* list operations *)
append  : list('a) * list('a) -> list('a)
reverse  : list('a) -> list('a)
length  : list('a) -> nat
hd  : list('a) -> 1 + 'a
tl  : list('a) -> 1 + list('a)
```

B.2 Integers

See the file EXAMPLES/INT.ch.

```
(* constructors *)
d0  : 1 -> digit
d1  : 1 -> digit
d2  : 1 -> digit
d3  : 1 -> digit
d4  : 1 -> digit
d5  : 1 -> digit
d6  : 1 -> digit
d7  : 1 -> digit
d8  : 1 -> digit
d9  : 1 -> digit
INT  : (1 + 1) * list(digit) -> int

(* digit operations *)
digit_2_nat  : digit -> nat
succ_digit  : digit -> bool * digit
pred_digit  : digit -> bool * digit
add_digit  : digit * digit -> digit * digit
sub_digit  : digit * digit -> digit * digit
mul_digit  : digit * digit -> digit * digit
max_digit  : digit * digit -> digit
min_digit  : digit * digit -> digit
eq_digit  : digit * digit -> bool
gt_digit  : digit * digit -> bool
le_digit  : digit * digit -> bool
ge_digit  : digit * digit -> bool
lt_digit  : digit * digit -> bool
nat_2_digit2  : nat -> bool * digit

(* list of digit (ie. digits) operations *)
compress_digits  : list(digit) -> list(digit)
make_length_digits  : nat -> list(digit)
make_same_length_digits  : list(digit) * list(digit) -> list(digit) * list(digit)
head_digits  : list(digit) -> digit
tail_digits  : list('a) -> list('a)
succ_digits  : list(digit) -> list(digit)
pred_digits  : list(digit) -> list(digit)
add_digits  : list(digit) * list(digit) -> list(digit)
sub_digits  : list(digit) * list(digit) -> list(digit)
cmp_digits {(digit * digit) * 'a -> 'b} : (list(digit) * list(digit)) * 'a -> 'b
digits_2_nat  : list(digit) -> nat
nat_2_digits  : nat -> list(digit)
folddigits {1 * 'a -> 'b, 'b * 'a -> 'b} : list(digit) * 'a -> 'b
mul_digits  : list(digit) * list(digit) -> list(digit)
div_mod_digits'  : list(digit) * list(digit) -> digit * list(digit)
div_mod_digits  : list(digit) * list(digit) -> 1 + (list(digit) * list(digit))

(* integer operations *)
succ_int  : int -> int
pred_int  : int -> int
add_int  : int * int -> int
mul_int  : int * int -> int
div_mod_int  : int * int -> 1 + (int * int)
div_int  : int * int -> 1 + int
mod_int  : int * int -> 1 + int
neg_int  : int -> int
sub_int  : int * int -> int
nat_2_int  : nat -> int
int_2_nat  : int -> nat
foldint {1 * 'a -> 'b, 'b * 'a -> 'b} : int * 'a -> 'b
pos_0_int  : int -> int
cmp_int {(digit * digit) * 'a -> 'b} : (('b * 'b) * (int * int)) * 'a -> 'b
gt_int  : int * int -> bool
ge_int  : int * int -> bool
lt_int  : int * int -> bool
le_int  : int * int -> bool
eq_int  : int * int -> bool
```

B.3 Strings

See the file EXAMPLES/STRING.ch.

```
(* Constructors *)
CHAR  : list(digit) -> char
STRING  : list(char) -> string

(* character  operations *)
eq_char  : char * char -> bool

(* string operations *)
eq_string  : string * string -> bool
length_string  : string -> nat
substring  : string * (nat * nat) -> string
ord_string  : string -> list(digit)
chr_string  : list(digit) -> string
```