Core-Haskell

Reference Manual

Joseph Manning

Department of Computer Science

University College Cork

September 2013

Core-Haskell is a very small subset of the programming language Haskell.

Yet it is remarkably powerful and contains many of the key ideas of the full language.

Its purpose is to reveal the essence of functional programming in a simplified setting.

It was designed and implemented at UCC by Haodong Guo and Joseph Manning.

Throughout this manual, the term 'Core-Haskell' will be written as 'C-H'.

PROGRAMS

A C-H program is composed of a sequence of **expressions** and/or **definitions**. Each of these expressions is evaluated, and its value is written out; each definition attaches a name to an item.

ITEMS and EXPRESSIONS

The atomic data entities occurring in C-H are called **items**. There are four types of items in C-H :

numbers, booleans, lists, functions.

Items are denoted by means of **expressions**. For example, each of the expressions `2+3`, `5`, `9-4` denotes the same item, the number `5`. Every expression is simply a means of denoting an item, and it may always be replaced by any other expression which denotes the same item.

Apart from processing definitions, the C-H interpreter is just an expression simplifier; it reads in expressions, simplifies them, and then writes out the items which they denote. Thus, for example, the input `2+3` produces the output `5`.

NUMBERS

A **number** is an integer, written as a sequence of decimal digits, possibly preceded by a '−' sign

e.g. `5`, `0`, `-28`, `4371`

The arithmetic operators

`+` (infix) `-` (prefix, infix) `*` (infix)

and the arithmetic functions

`div` (prefix) `mod` (prefix)

generate number items from number items. For example,

`17+3` ⇒ `20`, `17-3` ⇒ `14`, `17*3` ⇒ `51`, `div 17 3` ⇒ `5`, `mod 17 3` ⇒ `2`

BOOLEANS

A **boolean** is one of the two values `True` and `False`.

The boolean operators

`&&` (and; infix) `||` (or; infix)

and the boolean function

`not` (prefix)

generate boolean items from boolean items.

LISTS

A **list** is an ordered sequence of items called its **components**. A list is either of the form

`[]` (the empty list)

or

`h : t` (`h` an item, `t` a list).

Colon (`:`) is an infix operator whose left operand `h` is an item and whose right operand `t` is a list, and which produces a new list one component longer by attaching the item to the front of the list.

Examples of lists are

```
[]                                                    ( 0 components )
5 : []                                                ( 1 component )
[] : [] : []                                          ( 2 components )
( 1 : 2 : [] ) : ( 3 : 4 : 5 : [] ) : []              ( 2 components )
( not True ) : ( False || True ) : ( 2*4 < 7 ) : []   ( 3 components )
```

In Haskell, all components of a list must have the same type; however, C-H does not check this.

The functions `head` and `tail` are defined for non-empty lists by

```
head ( h : t ) ⇒ h
tail ( h : t ) ⇒ t
```

For example,

```
head ( 1 : 2 : [] ) ⇒ 1
tail ( 1 : 2 : [] ) ⇒ 2 : []
head ( head ( tail ( ( 1 : 2 : [] ) : ( 3 : 4 : [] ) : [] ) ) ) ⇒ 3
```

The functions `head` and `tail` may not be applied to the empty list.

FUNCTIONS

A **function** is a correspondence from an item (called its **argument**) to an item (called its **result**). Functions are written in the form

\ <*NAME*> -> <*EXPRESSION*> (\ denotes the symbol λ and is pronounced *lambda*)

where <*NAME*> represents an arbitrary argument and <*EXPRESSION*> specifies the result in terms of <*NAME*>, i.e. the item to which it is mapped. For example,

```
\ n -> n * n
```

is a function which expects a number item as argument and returns its square as result.

Function application is denoted simply by writing the function followed by its argument, i.e.

(\ <*NAME*> -> <*EXPRESSION*>) <*ARGUMENT*>

and its result is obtained by evaluating <*EXPRESSION*> with each occurrence of <*NAME*> replaced by <*ARGUMENT*> (the parentheses here are required for precedence, and are not an intrinsic part of the function application itself). Thus

```
( \ n -> n * n ) 2 ⇒ 2 * 2 ⇒ 4 .
```

Every function takes a single item as argument and returns a single item as result. Both the argument and the result may be items of *any* type.

In particular, the result of a function may itself be a function. Such function-generating functions (called *curried functions*) provide a way of representing multi-argument functions. For example, the function

```
\ n1 -> \ n2 -> n1 + n2
```

when applied to an argument a_1 returns the function `\ n2 -> ` a_1 `+ n2`, which when applied, in turn, to an argument a_2 returns the sum $a_1 + a_2$ of the *two* arguments. Thus

```
( \ n1 -> \ n2 -> n1 + n2 ) 2 3 ⇒ ( \ n2 -> 2 + n2 ) 3 ⇒ 2 + 3 ⇒ 5 .
```

RELATIONAL OPERATORS

The **relational operators** == and /= take two items of the same type as operands and generate the boolean item `True` or `False` according as these items are equal or unequal, respectively. For example, each of the following expressions has the value `True`:

```
1 + 1  ==  2

not True  /=  True

1 : 2 : []  ==  8 - 7 : 1 + 1 : []

1 : 2 : []  /=  2 : 1 : []
```

Two lists are considered equal iff they have the same number of components and corresponding components are equal.

Testing the equality of functions is a computationally unsolvable problem, so comparing functions using == or /= is *not* allowed.

The relational operators < <= > >= can be used to compare *numbers*.

CONDITIONAL EXPRESSIONS

A **conditional expression** is a way of selecting between the values of two expressions, based on some boolean condition. It is an *expression*, of the form

`if` <*CONDITION*> `then` <*EXPRESSION-1*> `else` <*EXPRESSION-2*>

and its overall value is the value of either <*EXPRESSION-1*> or <*EXPRESSION-2*> depending on whether the boolean expression <*CONDITION*> is `True` or `False`, respectively.

For example,

```
\ n -> if n >= 0 then n else -n
```
(this function returns the absolute value of its argument)

```
\ n -> if n > 0 then 1 else if n < 0 then -1 else 0
```
(for multi-way branching, use another conditional expression for <*EXPRESSION-2*>)

DEFINITIONS

A **definition** is a means of attaching a *name* to an item. It has the form

<*NAME*> = <*EXPRESSION*>

and subsequently, <*NAME*> denotes the item denoted by <*EXPRESSION*>. For example,

```
hoursperweek = 24 * 7
```

causes `hoursperweek` to denote the number item `168`.

Only a single definition may ever be supplied for any given <*NAME*>.

FILES

Definitions, and nothing else, must be placed inside *files*, which must have the extension '`.hs`'. To read and process the definitions in a given file, issue the top-level command

`:load` <*FILENAME*>

For this command, the extension '`.hs`' may be omitted from <*FILENAME*>.

LAZY EVALUATION

An important and powerful aspect of C-H is its use of *lazy evaluation*: operations are not performed unless/until their results are actually needed. This has several significant consequences:

Call-by-Need: When a function is applied to an argument, the argument remains unevaluated until its value is needed within the function. In particular, if its value is never needed, it is never evaluated; thus, `( \ n -> 2 + 3 ) ( div 1 0 )` evaluates to 5, rather than giving an error.

Short-Circuit Boolean Operators: Unless its value is needed, the second operand of `&&` or `||` is not evaluated; thus

if *P* evaluates to `False` then *P* `&&` *Q* is known to be `False`

if *P* evaluates to `True` then *P* `||` *Q* is known to be `True`

and in each case *Q* need not, and will not, be evaluated.

Lazy Definitions: When a definition is processed, its <*EXPRESSION*> is not actually *evaluated*, and thus can contain occurrences of <*NAME*>s which have yet to be defined. This allows:

Recursive Definitions:

```
factorial = \ n -> if n == 0 then 1 else n * factorial ( n - 1 )
```

Forward Definitions:

```
a = b + 1
b = 4
```

Mutually Recursive Definitions:

```
iseven = \ n -> n == 0 || isodd  ( n - 1 )
isodd  = \ n -> n /= 0 && iseven ( n - 1 )
```

Infinite Lists: The `:` operator is lazy, which allows infinite lists to be specified and manipulated with ease in C-H. For example,

```
ones = 1 : ones
```

defines the infinite list `1 : 1 : 1 : 1 : ...`, while

```
from = \ n -> n : from ( n + 1 )
```

results in `from 1` being the infinite list `1 : 2 : 3 : 4 : 5 : ...`.

Entire infinite lists are never actually *constructed*; instead, their components are produced only upon demand, with the list being expanded just as far as is strictly necessary. For example,

```
head ( tail ( from 1 ) )
⇒ head ( tail ( 1 : from ( 1 + 1 ) ) )
⇒ head ( from ( 1 + 1 ) )
⇒ head ( ( 1 + 1 ) : from ( ( 1 + 1 ) + 1 ) )
⇒ 1 + 1
⇒ 2
```

FURTHER DETAILS

Operator Precedence: The operators of C-H, in decreasing order of precedence, are as follows:

Function Application (note that `div`, `mod`, `not`, `head`, `tail` are functions)

`*`

`+ -`

`:`

`== /= < <= > >=`

`&&`

`||`

In evaluating a multi-operator expression, operators of higher precedence are applied before those of lower precedence. Operators of equal precedence are applied from left-to-right, apart from `:` which is applied from right-to-left. However, parentheses may be used to override precedence and explicitly control the order of application of operators; this is the *only* use of parentheses in C-H.

Syntax of a *<NAME>*: A *<NAME>* must start with a *lower*-case letter, and can continue with any sequence of lower-case letters, upper-case letters, digits, or the characters '`'` or '`_`'. The words `if`, `then`, `else` are reserved and cannot be used as *<NAME>*s.

Scope: The scope of the *<NAME>* in a *function* is the associated *<EXPRESSION>*, apart from any nested functions in which that *<NAME>* is re-used; the scope of the *<NAME>* in a *definition* is the entire program, apart from any functions in which that *<NAME>* is re-used. For example, with the definition

```
n = 1
```

the following top-level expression has the value `12` (`= 1 + 3 + 2 * 2 + 3 + 1`):

```
n + ( \ n -> ( n + ( \ n -> n * n ) 2 + n ) ) 3 + n
```

Expressions Evaluated Once: A *<NAME>* becomes bound to an *<EXPRESSION>* during either a function application or a definition. Under lazy evaluation, when that *<NAME>* is first used, the *<EXPRESSION>* is evaluated; at that point, *all* occurrences of the *<NAME>* are re-bound to the *value* of the *<EXPRESSION>*. Thus, for example, in the function application

```
( \ n -> n * n ) ( 2 + 3 )
```

the expression `2+3` is evaluated only *once*; likewise, with the definition

```
hoursperweek = 24 * 7
```

the expression `24 * 7` will be evaluated only the *first* time that `hoursperweek` is used.

Format of Output: Output is produced when a top-level expression is evaluated.

A **number** item is written out in standard decimal form.

A **boolean** item is written out as `True` or `False`.

A **list** item is written out by writing out each component, with sublists in parentheses, separated by `:` symbols and terminated by `[]`.

A **function** item is simply written out as `<FUNCTION>`.

Comments: The symbol `--` introduces a *comment*, and all further text on that line is ignored.

Code Layout: Blanks, tabs, and newlines are called *whitespace* characters. When writing C-H, whitespace may be used freely to separate tokens. C-H does not strictly enforce the 'offside rule' of Haskell; however, in multi-line definitions, lines after the first one must be indented.

Running the C-H Interpreter: The C-H Interpreter is an interactive terminal-based program, invoked by the command `ch` (in Linux). At startup, it automatically reads and processes a file of standard definitions, and then repeatedly accepts three types of instructions from the terminal:

<*EXPRESSION*>

evaluate the expression and output the result.

`:load` <*FILENAME*>

read and process the file <*FILENAME*>, which can contain only definitions.

<*END-OF-FILE*> (Control-D in Linux)

terminate the execution of the Interpreter.

C-H SYNTAX

{ *X* }* denotes zero or more occurrences of *X*
[*X*] denotes zero or one occurrences of *X*
X | *Y* denotes either *X* or *Y*

<*SESSION*> ::= { <*EXPR-1*> | `:load` <*FILENAME*> }* <*END-OF-FILE*>

<*EXPR-1*> ::= `\` <*NAME*> `->` <*EXPR-1*> |
`if` <*EXPR-2*> `then` <*EXPR-1*> `else` <*EXPR-1*> |
<*EXPR-2*>

<*EXPR-2*> ::= <*EXPR-3*> { `||` <*EXPR-3*> }*

<*EXPR-3*> ::= <*EXPR-4*> { `&&` <*EXPR-4*> }*

<*EXPR-4*> ::= <*EXPR-5*> [<*RELOP*> <*EXPR-5*>]

<*EXPR-5*> ::= <*EXPR-6*> { `:` <*EXPR-6*> }*

<*EXPR-6*> ::= [`-`] <*EXPR-7*> { <*ADDOP*> <*EXPR-7*> }*

<*EXPR-7*> ::= <*EXPR-8*> { `*` <*EXPR-8*> }*

<*EXPR-8*> ::= <*EXPR-9*> { <*EXPR-9*> }*

<*EXPR-9*> ::= <*NAME*> | <*NUMBER*> | `True` | `False` | `[]` | `(` <*EXPR-1*> `)`

<*RELOP*> ::= `==` | `/=` | `<` | `<=` | `>` | `>=`

<*ADDOP*> ::= `+` | `-`

<*FILENAME*> ::= <*NAME*>`.hs`

<*END-OF-FILE*> ::= `Control-D` (in Linux)

<*NAME*> ::= <*LC-LETTER*> { <*LC-LETTER*> | <*UC-LETTER*> | <*DIGIT*> | `'` | `_` }*

<*NUMBER*> ::= [`-`] <*DIGIT*> { <*DIGIT*> }*

<*LC-LETTER*> ::= `a` | `b` | `c` | ... | `z`

<*UC-LETTER*> ::= `A` | `B` | `C` | ... | `Z`

<*DIGIT*> ::= `0` | `1` | `2` | ... | `9`

<*DEFN*> ::= <*NAME*> `=` <*EXPR-1*>