Haskell Computer Algebra System

Rob Tougher

December 15, 2007

Outline

- Tutorial
- Implementation
- Looking Back

Tutorial: Language Summary

HCAS is a subset of Haskell, plus support for computer algebra.

- Purely functional language
- Construction of mathematical expressions
- Navigation of mathematical expressions

Tutorial: Running HCAS

```
$ echo "main = 7" | ./hcasi
7
$
```

Tutorial: Hello World!

The HCAS Hello World program:

```
main = "Hello World!"
```

Output: “Hello World!”

Tutorial: Basic Data Types

- Number – integer and floating point types for numbers
- Character – single printable character
- List – contains zero or more elements
- String – list of characters

Tutorial: Numbers

Numbers represent integers or floating point types:

```
main = 7.5
```

Output: 7.5

Tutorial: Strings

Strings represent a list of characters:

```
main = "Hello World!"
```

Output: “Hello World!”

Tutorial: Lists

Lists represent zero or more items:

```
main = [1,2,3,4,5]
```

Output: [1,2,3,4,5]

Tutorial: Operators

- Math operators – addition, subtraction, multiplication, etc. For basic math.
- List operators – the "++" operator concatenates two lists.

Tutorial: Math Operators

Math operators follow normal rules of associativity and precedence:

```
main = 2 + 3 * 4
```

Output: 14

Tutorial: List Operators

The concatenation operator lets you concatenate two lists:

```
main = [1,2,3] ++ [4,5]
```

Output: [1,2,3,4,5]

Tutorial: Functions

Functions represent callable HCAS expressions:

- Zero or more input arguments.
- Applicative-order evaluation.
- Strict evaluation

Tutorial: Calling a Function, No Arguments

Calling a function with zero arguments:

```
foo = 7
main = foo
```

Output: 7

Tutorial: Calling a Function, w/ Arguments

Calling a function with one or more arguments:

```
add(x, y) = x + y
main = add(3,4)
```

Output: 7

Tutorial: Function List Patterns

The colon operator in a function argument creates a list pattern:

```
reverse(x:xs) = reverse(xs) ++ [x]
reverse([]) = []
main = reverse("Hello World!")
```

Output: “!dlroW olleH”

Tutorial: Math Expression Data Type

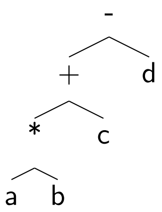
If an identifier does not match a function name, it represents a mathematical expression:

```
main = x + y
```

Output: $x + y$

A math expression is stored as a tree, using the normal rules of precedence and associativity:

```
main = a*b + c - d
```

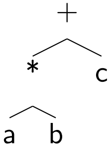


Tutorial: Function Math Patterns

You can put any math operators in a function argument. These create math patterns:

```
printType(x+y) = "addition"
printType(x-y) = "subtraction"
main = printType(a*b+c)
```

Output: “addition”
(In the call to printType, x refers to “a*b” and y refers to “c”.)



Tutorial: Let Expressions

Let expressions create a new scope:

```
main =
    let
        x = 7
        y = 8
        add(a,b) = a+b
    in
        add(x,y)
```

Output: 15

Tutorial: Derivative Example

```
main = derivative(3*x^2+2*x)

derivative(a+b) = derivative(a) + derivative(b)
derivative(a-b) = derivative(a) - derivative(b)
derivative(c*x^e) = c*e*simplify(x^(e-1))
derivative(c*x) = c
derivative(x) = 0

simplify(x^1) = x
simplify(x^0) = 1
simplify(x+0) = x
simplify(0+x) = x
simplify(x+y) = simplify(x) + simplify(y)
simplify(x-y) = simplify(x) - simplify(y)
simplify(x) = x
```

Output: 6*x+2

Tutorial: Questions?

Any questions on the language?

Implementation: Technologies

- Haskell – the entire interpreter is written in Haskell, using the Glasgow Haskell Compiler, v 6.6.1.
- HUnit – a unit testing framework, similar to JUnit and NUnit.
- Parsec – a monadic parsing library for top-down parsing.

Implementation: Haskell Modules

- AST.hs – contains the abstract syntax tree.
- Parser.hs – contains the parsing code. Takes an input string, and returns an AST.
- Interpreter.hs – contains the interpreter code.
- MainInterpreter.hs – contains the main bootup code (reading from stdin, writing to stdout).

Implementation: AST.hs

```
data Block =  Block [Statement]
data Statement = Function String [Expression] Expression
data Expression =
    -- Strings and lists.
    List [Expression]
    | Concat Expression Expression
    | ListPattern [Expression]
    | CharValue Char

    -- Function-related items
    | Call String [Expression]
    | Let Block Expression
    ...
```

Implementation: Parser.hs

```
identifier :: Parser String
identifier =
    do {
        c <- letter;
        cs <- many (identifierChar);
        return (c:cs);
    }

identifierChar =
    do {
        (alphaNum <|> char '_');
    }
```

Implementation: Interpreter.hs

```
interpret :: [Block] -> Expression -> Expression

interpret _ (Number n) = (Number n)

interpret blocks (Let block expr) =
    (interpret ([block] ++ blocks) expr)

interpret blocks (Addition left right) =
        (addition left' right')
    where
        left' = (interpret blocks left)
        right' = (interpret blocks right)
        addition (Number n1) (Number n2) = (Number (n1 + n2))
        addition left'' right'' = (Addition left'' right'')
```

Implementation: MainInterpreter.hs

```
main =
    do {
        script <- getContents;
        case (parse file "" script) of
            (Right parsed) ->
                do {
                    interpreted <- return (interpretFile parsed);
                    putStrLn (showHCAS interpreted);
                }
            (Left err) ->
                do {
                    putStrLn (show err);
                }
        }
```

Implementation: Unit Testing

Unit testing used to verify functionality. Three types of tests:

- Haskell unit tests
- HCAS boolean unit tests
- HCAS expected vs. actual unit tests

Implementation: Haskell Unit Tests

Haskell unit tests are writing using Haskell:

```
testNum2 = TestCase (
    do {
        expected <- return (Number 1.3);
        (Right actual) <- return (parse numberAtom "" "1.3");
        assertEqual "testNum2" expected actual;
        }
    )
```

Implementation: HCAS Boolean Unit Tests

HCAS boolean tests are HCAS scripts that must return a boolean true value:

```
main = 7 == 1 + 2 + 4
```

Output: True

Implementation: HCAS Expected vs. Actual Unit Tests

HCAS expected vs. actual tests have an HCAS script and expected output file for each test:

```
addition.hcas
addition_expected.txt
subtraction.hcas
subtraction_expected.txt
functioncall.hcas
functioncall_expected.txt
...
```

Implementation: Questions?

Any questions on the implementation?

Looking Back

- Haskell works well for parsing. Parsec is fun.
- Professor is right – get started early.
- I wish I wrote a compiler (instead of an interpreter). I missed out on generation of IR and assembly code.
- If I had more time, I would add static typing.

Questions?

Any final questions?