Bernie Pope, bjpope@unimelb.edu.au

Implementing Python in Haskell, twice!

Melbourne Python Users Group, Monday 7 July 2014

Is this about Haskell or Python?

It's a bit of both

Overview

- Motivation
- Parsing Python
- Translating Python to Haskell
- Bytecode compilation and interpretation
- Future directions

Motivation

- A fun way to kill time.
- Haskell is particularly good for writing compilers.
- Python is a (mostly) simple language to implement.

Lexical analysis

- I use Alex (like Lex/Flex) for lexical analysis.
- Lexical analysis takes the input source program text, the filename (for error messages), and breaks the source up into a sequence of tokens:

```
lex :: String -> Filename -> Either ParseError [Token]
```

- Python's lexical structure is well defined:

```
https://docs.python.org/3/reference/lexical_analysis.html
```

Parsing

- I use Happy (like Yacc/Bison) for parsing.
- Parsing takes the input source program text, the filename (for error messages), and builds an abstract syntax tree:

```
parseModule :: String -> Filename
            -> Either ParseError (ModuleSpan, [Token])
```

- Python's grammar is well defined:

`https://docs.python.org/3/reference/grammar.html`

Parsing

- Internally the parser calls the lexer and processes the generated sequence of tokens.

language-python

- Eventually I'd written a lexer, parser and pretty printer.
- Support for both Python 2.x and 3.x.
- Could parse the CPython test suite.
- The result is a library called language-python:

  `https://github.com/bjpop/language-python`
- Now what?

berp: translating Python to Haskell

- Thought experiment: what would it take to translate Python into Haskell?
- Okay, what is the semantics of Python?

```
while True:
    try:
        1/0
    except:
        break
    finally:
        continue
```

berp: translating Python to Haskell

- Thought experiment: what would it take to translate Python into Haskell?
- Okay, what is the semantics of Python?

```
while True:
    try:
        1/0
    except:
        break
    finally:
        continue
```

> SyntaxError: 'continue' not supported inside 'finally' clause

berp: translating Python to Haskell

- The real trick is in encoding Python's imperative effects into a pure language (state, control flow, mutable values, input/output).
- Haskell's monad (transformers) provide an elegant way to combine different effects together.

berp: translating Python to Haskell

- An example, recursive factorial using an accumulator:

```
def fac(n, acc):
    if n == 0:
        return acc
    else:
        return fac(n - 1, n * acc)
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

```
def fac(n, acc):
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

```
n == 0:
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

return acc

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

```
return fac(n - 1, n * acc)
```

berp: translating Python to Haskell

- translated into Haskell by berp:

```
def _s_fac 2 none
    (\ [_s_n, _s_acc] ->
        ifThenElse
            (do _t_6 <- read _s_n
                _t_6 == 0)
            (do _t_7 <- read _s_acc
                ret _t_7)
            (do _t_0 <- read _s_fac
                _t_1 <- read _s_n
                _t_2 <- _t_1 - 1
                _t_3 <- read _s_n
                _t_4 <- read _s_acc
                _t_5 <- _t_3 * _t_4
                tailCall _t_0 [_t_2, _t_5]))
```

berp: translating Python to Haskell

- Berp has some cute party tricks:
    - Tail call optimisation, `fac` runs in constant stack space.
    - `callCC` (call with current continuation, borrowed from Scheme).

berp: translating Python to Haskell

- callCC example in berp:

```
>>> def f():
...     count = 0
...     k = callCC(lambda x: x)
...     print(count)
...     if count < 3:
...         count = count + 1
...         k(k)
...
>>> f()
0
1
2
3
```

Is Haskell a good target for Python compilation?

- Pros:
  - We get to use the Haskell runtime features for free: garbage collection, threads, I/O.
- Cons:
  - The runtime representation of Python state (in berp) is pretty heavy weight (slow).
  - Python uses a lot of mutation. Haskell compilers are not optimised for this.

blip: a bytecode compiler and interpreter

- Having pursued the berp thought experiment far enough I decided to try making a bytecode compiler and interpreter.
- I started by writing a program to read .pyc files generated by CPython. Then used it to reverse engineer the meaning of the bytecode.

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

```
n == 0
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

```
return acc
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

```
return fac(n - 1, n * acc)
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

`return None`

blip: a bytecode compiler and interpreter

- Bytecode for the factorial function:

```
0 LOAD_FAST 0
3 LOAD_CONST 1
6 COMPARE_OP 2
9 POP_JUMP_IF_FALSE 19
12 LOAD_FAST 1
15 RETURN_VALUE
16 JUMP_FORWARD 21
19 LOAD_GLOBAL 0
22 LOAD_FAST 0
25 LOAD_CONST 2
28 BINARY_SUBTRACT
29 LOAD_FAST 0
32 LOAD_FAST 1
35 BINARY_MULTIPLY
36 CALL_FUNCTION 2
39 RETURN_VALUE
40 LOAD_CONST 0
43 RETURN_VALUE
```

`return None`

dead code

blip: a bytecode compiler and interpreter

- It turns out that CPython uses a very straightforward compilation scheme. Easy to emulate in Haskell:

```
compileExpr :: ExprSpan -> Compile ()
compileExpr (CondExpr {..}) = do
    compile ce_condition
    falseLabel <- newLabel
    emitCodeArg POP_JUMP_IF_FALSE falseLabel
    compile ce_true_branch
    restLabel <- newLabel
    emitCodeArg JUMP_FORWARD restLabel
    labelNextInstruction falseLabel
    compile ce_false_branch
    labelNextInstruction restLabel
```

blip: a bytecode compiler and interpreter

- Blip generates bytecode which is compatible with CPython.
- I originally used CPython to test the generated byte code.
- Then I decided to write a bytecode interpreter in Haskell too.

blip: a bytecode compiler and interpreter

- Evaluating the bytecode is reasonably straightforward:

```
evalOneOpCode :: HeapObject -> Opcode -> Word16 -> Eval ()
evalOneOpCode (CodeObject {..}) opcode arg =
   case opcode of
      CALL_FUNCTION -> do
         functionArgs <- replicateM (fromIntegral arg)
                                   popValueStack
         functionObjectID <- popValueStack
         functionObject <- lookupHeap functionObjectID
         callFunction functionObject $ reverse functionArgs
      JUMP_ABSOLUTE -> setProgramCounter $ fromIntegral arg
      … etcetera …
```

blip: a bytecode compiler and interpreter

- I've implemented about half of the bytecode instructions so far. Many of them are simple manipulations of the stack.
- However, implementing attribute lookup completely and correctly is quite difficult.

blip: a bytecode compiler and interpreter

- Time for a little demo:

```
$ blip
Berp version 0.2.1, type control-d to exit.
>>> def fac(n):
...     if n <= 1:
...         return 1
...     else:
...         return fac(n - 1) + fac(n - 2)
...
>>> x = 0
>>> while x < 10:
...     print(fac(x))
...     x = x + 1
...
1
1
2
3
5
8
13
21
34
55
```

Future directions

- Complete the byte code interpreter
- Add language extensions:
    - Tail call optimisation?
    - Algabraic types, pattern matching?
- Possibly optimise execution by compiling to machine code.

Source code

- https://github.com/bjpop/language-python
- https://github.com/bjpop/berp
- https://github.com/bjpop/blip