Implementing Python in Haskell, twice.

Bernie Pope
Melbourne Haskell Users Group
24th April 2014

Overview

- How it all started
- language-python
- berp
- blip
- What’s the point and where will it end?

How it all started.

In 2009 I was teaching Advanced Functional Programming to honours and masters students.

I needed a project which covered parser combinators and continuations, amongst other things.

Decided to get the students to implement a small imperative programming language.

How it all started.

At the same time I was teaching Python to first year students.

My brain merged the two things together.

The small imperative language project inherited a lot of Pythonic features.

How it all started

I wrote a sample solution to give to students.

It was fun.

I started wondering: how hard would it be to extend my sample solution to cover all of Python?

First things first

I had a toy parser written in Parsec, but it was slow and incomplete.

Decided to rewrite it (properly) using Alex and Happy.

language-python

Python’s official grammar is LL(1); easy to parse in recursive descent.

Happy supports LALR(1) (and extensions).

Initially I tried to generate the parser from the grammar, but I found it easier to write by hand.

Happy parser generator

Grammar rule:

```
funcdef: 'def' NAME parameters ['->' test] ':' suite
```

Corresponding Happy parser rule:

```
funcdef :: { StatementSpan }
funcdef
   : 'def' NAME parameters opt(right('->',test)) ':'
         suite
     { makeFun $1 $2 $3 $4 $6 }
```

Happy's parameterized productions

```
left(p,q): p q { $1 }

right(p,q): p q { $2 }

opt(p)
   :    { Nothing }
   | p  { Just $1 }

many0(p)
   : many1(p) { $1 }
   |          { [] }
```

Tricky syntactic issues

Indentation/dedentation is handled by the lexer.

Comments are maintained in the output, but currently separate from the AST. Where to put them? Important for refactoring tools.

Source locations are kept as annotations in the AST; but troublesome to use the AST for other purposes.

Some loose ends

Unicode support is patchy.

Uses Prelude.String; should probably use Data.Text.

Separate package for testing, language-python-test, but needs lots of work.

Testing parsers (properly) is frustrating.

Now what?

I had a parser, but now what to do?

I could write an interpreter (AST walker), but that would be really slow.

Someone once told me: if you can write an interpreter then it is nearly as easy to write a translator.

The berp thought experiment

What would it take to translate Python into Haskell?

Fortunately public transport gave me a lot of thinking time.

In the great tradition of writing compilers I wrote the first translations by hand. It seemed feasible.

Okay, what is Python's semantics?

Control flow can be tricky:

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

What is this supposed to do?

Okay, what is Python's semantics?

Let's ask Python:

```
python foo.py
  File "foo.py", line 7
    continue
SyntaxError: 'continue' not supported inside
'finally' clause
```

Python's semantics is "whatever CPython does", but see: An executable operational semantics for Python, http://gideon.smdng.nl/wp-content/uploads/thesis.pdf

Don’t worry about semantics, get hacking

In 2010 I had to fly to the USA.

I took my laptop on the plane.

By the end of the trip I had a workable Python-to-Haskell translator.

Key types

```
type Eval a = StateT EvalState (ContT Object IO) a
type ObjectRef = IORef Object
data Object
   = Object
     { object_identity :: !Identity
     , object_type :: !Object
     , object_dict :: !Object
     }
     | etcetera

data EvalState =
   EvalState
   { state_control_stack :: !ControlStack
   , etcetera
   }
```

Example program

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

print(fac(1000, 1))
```

Example translated program

```
module Main where
import Berp.Base
import qualified Prelude
main = runStmt init
init
  = do _s_fac <- var "fac"
       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]))
       _t_8 <- read _s_print
       _t_9 <- read _s_fac
       _t_10 <- _t_9 @@ [1000, 1]
       _t_8 @@ [_t_10]
```

Example translated program

```
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]))
```

Party trick, how could I resist callCC?

```
>>> 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 compiling Python?

Pros:

- GHC's runtime features for free; GC, threads; I/O.

Cons:

- Runtime representation of Python state is heavy weight (i.e. slow).
- Python uses lots of mutation; Haskell is not good at this.

The honeymoon is over

I implemented a fair bit of the standard types, made a REPL, then shelved the project.

I’ve pursued that thought experiment far enough.

Back to the drawing board.

Take two; a bytecode compiler

During the implementation of berp I found it necessary to poke around in the CPython source.

This started a new train of thought: maybe I should write a bytecode compiler instead?

I’d never done that before, so I thought it might be educational.

Another flight to the USA

In 2012 I was flying back to the USA.

An ideal chance to start my new project.

With the help of GDB I managed to figure out Python’s bytecode representation.

I wrote a bytecode parser/pretty printer on that trip.

Bytecode for the factorial example

```
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
```

The compiler is straightfoward

```
newtype Compile a
   = Compile (StateT CompileState IO a)
   deriving (Monad, Functor, MonadIO, Applicative)

class Compilable a where
   type CompileResult a :: *
   compile :: a -> Compile (CompileResult a)
```

The compiler is straightfoward

```
-- compile the body of a function

instance Compilable Body where
   type CompileResult Body = PyObject
   compile (Body stmts) = do
      mapM_ compile stmts
      returnNone
      assemble
      makeObject
```

The compiler is straightfoward

```
compileExpr (AST.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
```

Testing the compiler

Test suite contains about 150 feature tests.

I use shelltestrunner to run tests and report results.

I also run the compiler over the CPython test suite.

What about a bytecode interpreter?

Over Christmas I had some spare time while visiting family overseas.

I thought about writing an operational semantics (on paper) for Python bytecode.

Then I came to my senses and started writing it in Haskell

The interpreter is straightforward

```
data EvalState =
   EvalState
   { evalState_objectID :: !ObjectID
   , evalState_heap :: !Heap
   , evalState_globals :: !Globals
   , evalState_frameStack :: ![HeapObject]
   }

newtype Eval a
   = Eval (StateT EvalState IO a)
   deriving (Monad, Functor, MonadIO, Applicative)
```

The interpreter is straightforward

```
data HeapObject
  = CodeObject
    { codeObject_code :: !ObjectID
    , codeObject_consts :: !ObjectID
    , codeObject_names :: !ObjectID
    , ... etcetera ...
    }
   | DictObject
     { dictHashTable :: !HashTable }
   | ... etcetera ...
```

The interpreter is straightforward

```
evalOneOpCode :: HeapObject -> Opcode -> Word16 -> Eval ()
evalOneOpCode (CodeObject {..}) opcode arg =
   case opcode of

      CALL_FUNCTION -> do
         functionArgs <- Monad.replicateM (fromIntegral arg) popValueStack
         functionObjectID <- popValueStack
         functionObject <- lookupHeap functionObjectID
         callFunction functionObject $ List.reverse functionArgs

      JUMP_ABSOLUTE -> setProgramCounter $ fromIntegral arg

      -- etcetera
```

How complete is the interpreter?

I’ve implemented about half of the bytecode instructions.

Python’s OOP implementation is quite tricky to get right.

I doubt many Python programmers know the full semantics of attribute resolution.

What now?

Can there be some practical benefit to all this effort?

Code repositories

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

https://github.com/bjpop/berp

https://github.com/bjpop/blip