Berp

An implementation of Python 3 in Haskell

Outline

- Demonstration
- Implementation details
- Novelties
- Tricky bits
- Future work

Demonstration

- The compiler in action.
- The interpreter in action.

Implementation

- Program translation: Python to Haskell.
- A monad for effects.
- Object representation.

Translation

```
def fac(n):
    result = 1
    while (n > 0):
       result = result * n
       n = n - 1
    return result
```

Translation

```
do _s_fac <- var "fac"
   def _s_fac 1 none
     (\ [_s_n] ->
        do _s_result <- var "result"
           _s_result =: 1
           while
             (do _t_0 <- read _s_n
                 _t_0 > 0)
             (do _t_1 <- read _s_result
                 _t_2 <- read _s_n
                 _t_3 <- _t_1 * _t_2
                 _s_result =: _t_3
                 _t_4 <- read _s_n
                 _t_5 <- _t_4 - 1
                 _s_n =: _t_5)
           _t_6 <- read _s_result
           ret _t_6)
```

Translation

```
do _s_fac <- var "fac"
   def _s_fac 1 none
     (\ [_s_n] ->
        do _s_result <- var "result"
           _s_result =: 1
           while
             (do _t_0 <- read _s_n
                 _t_0 > 0)
             (do _t_1 <- read _s_result
                 _t_2 <- read _s_n
                 _t_3 <- _t_1 * _t_2
                 _s_result =: _t_3
                 _t_4 <- read _s_n
                 _t_5 <- _t_4 - 1
                 _s_n =: _t_5)
           _t_6 <- read _s_result
           ret _t_6)
```

Python keywords become Haskell functions.

Translation

```
do _s_fac <- var "fac"
   def _s_fac 1 none
     (\ [_s_n] ->
        do _s_result <- var "result"
           _s_result =: 1
           while
             (do _t_0 <- read _s_n
                 _t_0 > 0)
             (do _t_1 <- read _s_result
                 _t_2 <- read _s_n
                 _t_3 <- _t_1 * _t_2
                 _s_result =: _t_3
                 _t_4 <- read _s_n
                 _t_5 <- _t_4 - 1
                 _s_n =: _t_5)
           _t_6 <- read _s_result
           ret _t_6)
```

Haskell operator syntax used where possible.

Translation

```
do _s_fac <- var "fac"
   def _s_fac 1 none
     (\ [_s_n] ->
        do _s_result <- var "result"
           _s_result =: 1
           while
             (do _t_0 <- read _s_n
                 _t_0 > 0)
             (do _t_1 <- read _s_result
                 _t_2 <- read _s_n
                 _t_3 <- _t_1 * _t_2
                 _s_result =: _t_3
                 _t_4 <- read _s_n
                 _t_5 <- _t_4 - 1
                 _s_n =: _t_5)
           _t_6 <- read _s_result
           ret _t_6)
```

Haskell literal syntax used where possible.

Translation

```
do _s_fac <- var "fac"
   def _s_fac 1 none
     (\ [_s_n] ->
        do _s_result <- var "result"
           _s_result =: 1
           while
             (do _t_0 <- read _s_n
                 _t_0 > 0)
             (do _t_1 <- read _s_result
                 _t_2 <- read _s_n
                 _t_3 <- _t_1 * _t_2
                 _s_result =: _t_3
                 _t_4 <- read _s_n
                 _t_5 <- _t_4 - 1
                 _s_n =: _t_5)
           _t_6 <- read _s_result
           ret _t_6)
```

Variables must be declared before use.

Some redundant reads to optimise away...

A monad for effects

```
type Eval a = StateT EvalState (ContT Object IO) a
```

```
data EvalState = EvalState { control_stack :: !ControlStack }
```

A monad for effects

```
type Eval a = StateT EvalState (ContT Object IO) a
```

```
data EvalState = EvalState { control_stack :: !ControlStack }
```

Python statements and expressions compile to Haskell terms of type:

`Eval Object`

A monad for effects

```
type Eval a = StateT EvalState (ContT Object IO) a
```

```
data EvalState = EvalState { control_stack :: !ControlStack }
```

State monad transformer provides a control stack.

A monad for effects

```
type Eval a = StateT EvalState (ContT Object IO) a
```

```
data EvalState = EvalState { control_stack :: !ControlStack }
```

Continuation monad transformer provides control flow (jumps).

A monad for effects

```
type Eval a = StateT EvalState (ContT Object IO) a
```

```
data EvalState = EvalState { control_stack :: !ControlStack }
```

IO monad provides input/output plus mutable variables, mutable arrays.

The control stack

```
data ControlStack

   = EmptyStack

   | ProcedureCall
     { procedure_return   :: Object -> Eval Object
     , control_stack_tail :: ControlStack
     }

   | ExceptionHandler
     { exception_handler  :: Maybe (Object -> Eval Object)
     , exception_finally  :: Maybe (Eval Object)
     , control_stack_tail :: ControlStack
     }

   ...
```

The joy of continuations

```
callCC :: (MonadCont m) => ((a -> m b) -> m a) -> m a
```

```
callProcedure :: Procedure -> [Object] -> Eval Object
callProcedure proc args =
   callCC $ \ret -> do
      push $ ProcedureCall ret
      proc args
```

```
ret :: Object -> Eval Object
ret obj = do
   stack <- unwind isProcedureCall
   procedure_return stack obj
```

The joy of continuations

```
callCC :: (MonadCont m) => ((a -> m b) -> m a) -> m a
```

```
callProcedure :: Procedure -> [Object] -> Eval Object
callProcedure proc args =
   callCC $ \ret -> do
      push $ ProcedureCall ret
      proc args
```

```
ret :: Object -> Eval Object
ret obj = do
   stack <- unwind isProcedureCall
   procedure_return stack obj
```

The joy of continuations

```
callCC :: (MonadCont m) => ((a -> m b) -> m a) -> m a
```

```
callProcedure :: Procedure -> [Object] -> Eval Object
callProcedure proc args =
   callCC $ \ret -> do
      push $ ProcedureCall ret
      proc args


ret :: Object -> Eval Object
ret obj = do
   stack <- unwind isProcedureCall
   procedure_return stack obj
```

Object representation

```
data Object
   = Object
     { object_identity :: !Identity
     , object_type     :: !Object
     , object_dict     :: !Object
     }
   | Type
     { object_identity    :: !Identity
     , object_type        :: Object
     , object_dict        :: !Object
     , object_bases       :: !Object
     , object_constructor :: !Procedure
     , object_type_name   :: !Object
     , object_mro         :: !Object
     }
   | Integer
     { object_identity :: !Identity
     , object_integer  :: !Integer
     }
   ...
   | None
```

Novelties

- Tail call optimisation demo.
- callCC as a Python primitive demo.

Tricky bits

```
def foo():
    yield 1
    yield 2
    yield 3

for x in foo():
    print(x)
```

Tricky bits

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

Future work

- Support the rest of the language:
  - Modules.
  - More standard library functions.
  - Dark corners, like metaclasses.
- Multi-threaded concurrency.
- Optimisations, performance tuning.