interpreter tweaks
mhe authored 1 hour ago

afad15cf

Interpreter.md 9.84 KB

Interpreter

```
module Interpreter where

import AbstractSyntax
```

We explain how to run programs written in abstract syntax:

```
                                | initial
                                | storage
                                v           final
       program tree       +-------+--------+  storage
     -------------------->|      run       | -------->
      (AbstractSyntax.hs) +----------------+
```

Representation of the storage

There are several ways to represent the storage:

1. As a list `m :: [(Identifier,Integer)]` describing a table associating integers to identifiers.
   - In this case, to find the value of an identifier `i`, we use the `lookup` function from the prelude:
     ```
     lookup i m
     ```
   - To update the storage `m` means to define a new *list* `m'` from the list `m`.
   - Unused program variables are simply not listed in `m`.
2. Again as a table but using hashing, or a tree or other efficient data structures.
3. As a function `m :: Identifier -> Integer`, again associating integers to identifiers.
   - In this case, to find the value of an identifier, we apply the function to the identifier.
     ```
     m i
     ```
   - To update the storage `m` means to define a new *function* `m'` from the function `m`.
   - Unused program variables are given the value `undefined` or `error`.

Here we adopt option 3:

```
type Storage = Identifier -> Integer
```

Empty storage

At the beginning, no storage location is initialized:

```
emptyStorage :: Storage
emptyStorage i = error ("Uninitialized variable " ++ i)
```

Updating the storage

To update a variable named `i` to the value `x` in `m :: Storage` means to produce a new `m' :: Storage` with `m' i = x` and `m' j = m j` for `j` distinct from `i`:

```
update :: Identifier -> Integer -> Storage -> Storage
update i x m = m'
 where
```

```haskell
    m' :: Storage
    m' j | i == j    = x
         | otherwise = m j
```

Booleans are represented as integers

We convert between numbers and booleans, so we only have the type of integers in our little language, like in the programming language `C`:

```haskell
number :: Bool -> Integer
number False = 0
number True  = 1

boolean :: Integer -> Bool
boolean 0 = False
boolean _ = True
```

Remark:

Notice that the above two functions don't form an isomorphism. We only have the equation `boolean(number b) = b` for all `b :: Bool`, or `boolean . number = id` where `id` is the identity function defined in the prelude, but not the other way round. Technically, one says that the above two functions exhibit the type `Bool` as a *retract* of the type `Integer`, with the function `boolean` the retraction and the function `number` the section. Then the type `Bool` is isomorphic to the set of fixed points of the function `number . boolean`, namely `0` and `1`. See also the section on retracts in the lecture on data types.

Evaluating expressions

We aim to evaluate expressions relative to a given storage:

An expression is either

- a constant (an integer), or
- a variable, or
- an operator together with a list of sub-expressions.

To evaluate, relative to storage `m :: Storage`,

- a **constant** means simply to extract the integer from it.
- a **variable**, say, `"x"`, simply means to look up the value, via `m "x"`.
- an **operator** and its sub-expressions, means to apply a Haskell operation associated to the operator to the evaluated sub-expressions. This is done in the following Haskell code, assuming that we evaluate operators through the auxiliary function `opEval` discussed below:

```haskell
eval :: Storage -> Expr ->  Integer
eval m (Constant x) = x
eval m (Var i)      = m i
eval m (Op o es)    = opEval o [eval m e | e <- es]
```

We look in detail at the last pattern, evaluating an operator `o` with sub-expressions `es` (a list of expressions, `[Expr]`).

Given an `OpName` (defined in the module AbstractSyntax) and the correct number of arguments (one or two here), we evaluate it:

```haskell
opEval :: OpName -> [Integer] -> Integer
opEval Add     [x, y] = x + y
opEval Sub     [x, y] = x - y
opEval Mul     [x, y] = x * y
opEval Div     [x, y] = x `div` y
opEval Mod     [x, y] = x `mod` y
opEval Eq      [x, y] = number(x == y)
opEval Leq     [x, y] = number(x <= y)
opEval Less    [x, y] = number(x <  y)
opEval Geq     [x, y] = number(x >= y)
opEval Greater [x, y] = number(x >  y)
opEval And     [x, y] = number(boolean x && boolean y)
opEval Or      [x, y] = number(boolean x || boolean y)
opEval Not     [x]    = number(not(boolean x))
opEval op      xs     = error ("Interpreter bug. "
                            ++ "Please contact the software maintainer. "
                            ++ "Tried to apply " ++ show op
```

```
                              ++ " to " ++ show xs)
```

Running programs

When we run a program we start with an initial storage and, if the program terminates, we get a final storage:

```
run :: Program -> Storage -> Storage
```

The assignment statement evaluates the expression `e` and stores it in the identifier `i` of the storage `m`:

```
run (i := e) m = update i (eval m e) m
```

To run an if-else-statement, we evaluate the expression `e`, and if its boolean value is true, we run the `then` branch `p` and otherwise we run the `else` branch `q`, with the given storage `m`:

```
run (IfElse e p q) m
    | boolean(eval m e) = run p m
    | otherwise         = run q m
```

Similarly, to run an if-statement, we evaluate the expression `e`, and if its boolean value is true, we run the `then` branch with the given storage `m`, and otherwise we leave the given storage unchanged:

```
run (If e p) m
    | boolean(eval m e) = run p m
    | otherwise         = m
```

To run a while-statement, we evaluate the condition `e`. If it is false, we leave the given storage `m` unchanged. If it is true, we run `p` on the storage `m` to get a new storage `m'`, which we use to recursively run the while-statement, to get a storage `m''`, which is the result, if this recursion terminates (it need not to):

```
run (While e p) m
    | boolean(eval m e) = m''
    | otherwise         = m
    where
      m'  = run p m
      m'' = run (While e p) m'
```

To run a block consisting of `n` program statements, starting from a storage `$m_1$`, we apply the first statement to get `$m_2$`, the second to get `$m_3$`, and so on, until we apply the last statement to get the final storage `m_n`. We express this recursively in the following definition, where the base case is when we have zero programs, in which no transformation to the storage takes place:

```
run (Block []) m = m

run (Block (p : ps)) m = m''
    where
      m'  = run p m
      m'' = run (Block ps) m'
```

An alternative approach to running programs

This approach shows that imperative programming is functional programming.

Now we define an alternative variant of the function `run :: Program -> Storage -> Storage`, using a different perspective. Specifically, we think of a program as a **storage transformer**, i.e., a map `Storage -> Storage`. Every program construction corresponds to a function that returns a storage transformer, i.e., that returns a function `Storage -> Storage`.

Firstly, the `assign`ment statement is a storage transformer with two parameters `i` and `e`:

```
assign :: Identifier -> Expr -> Storage -> Storage
assign i e = \m -> update i (eval m e) m
```

Next, the if-else-statement takes three functions as arguments, and produces one function as the result. The first argument is a property `p` of the storage. The second and third arguments are two storage transformers, and the result is a storage transformer.

```
ifElse :: (Storage -> Bool) -> (Storage -> Storage) -> (Storage -> Storage) -> (Storage -> Storage)
ifElse p f g = h
```

```
 where
   h m = if p m then f m else g m
```

Although we use `Storage` for interpreting imperative programs, this function makes sense for any type `s`:

```
ifElse :: (s -> Bool) -> (s -> s) -> (s -> s) -> (s -> s)
ifElse p f g = h
 where
   h m = if p m then f m else g m
```

Similarly, the functions below are defined with an arbitrary type `s` rather than the specific type `Storage`.

We don't need to consider an if-statement, because `ifElse p f id` does the job, where `id` is the identity function, defined in the prelude.

Next, the while-statement takes a property `p` of states, and a storage transformer `f`. The result is a storage transformer `g` that repeatedly applies `f` to a given storage `m` until it fails to satisfy `p`:

```
while :: (s -> Bool) -> (s -> s) -> (s -> s)
while p f = g
 where
   g m = if p m then g(f m) else m
```

The block-statement is simply the composition of finitely many functions: first apply the first function to the given storage, then the second function, etc., and finally the last function. For example, `block [f,g,h] m = h(g(f m))`:

```
block :: [s -> s] -> s -> s
block []     m = m
block (f:fs) m = block fs (f m)
```

Boolean expressions represent predicates of the storage:

```
booleanValue :: Expr -> (Storage -> Bool)
booleanValue e = \m -> boolean(eval m e)
```

Finally, with these definitions, the function `run` defined above is equivalent to the following:

```
run' :: Program -> Storage -> Storage
run' (i := e)       = assign i e
run' (IfElse e p q) = ifElse (booleanValue e) (run' p) (run' q)
run' (If e p)       = ifElse (booleanValue e) (run' p) id
run' (While e p)    = while  (booleanValue e) (run' p)
run' (Block ps)     = block  (map run' ps)
```

Next: The main program Runxy