Design Document for Project B

Gabe Dijkstra Hidde Verstoep

March 7, 2012

Language

We have extended the language with a `let`-construct and variables. It can be used as follows:

```
let stack = execute
              interpreter VirtualBox for i686-windows in i686-linux
            on
              platform i686-linux
            end
in
execute
  program Hello in i686-windows
on
  use stack
```

We also allow for constructions like the following:

```
execute
  execute
    compile
      program hello in Haskell
    with
      compiler ghc from Haskell to i586 in i686
    end
  on
    platform i586
  end
on
  platform i686
end
```

This allows us to also express diagrams with compilers that run on platforms.

Look in the `examples` directory for more examples and the `report.txt` to see which of those are supposed to not give any errors.

Transformation of the AST

To make typechecking and drawing easier we transform the AST `Diag` to a new data structure called `CoDiag` (short for "correct diagram"), which follows the visual structure more closely:

```
data CoDiag
  | CoDiag        pos :: SourcePos  d :: CoDiag_

data CoDiag_
  | CoProgram     p :: Ident   l :: Ident    d :: CoDiag
  | CoPlatform    m :: Ident
  | CoInterpreter i :: Ident   l :: Ident    m :: Ident
                  d :: CoDiag
  | CoCompiler    c :: Ident  l1 :: Ident   l2 :: Ident
                  m :: Ident  d1 :: CoDiag  d2 :: CoDiag
  | CoNothing
```

The idea of `Diag` is that `Program`, `Platform`, `Interpreter` and `Compiler` define inputs and/or outputs which are connected by `Execute` and/or `Compile`. The `CoDiag` structure does not have these connectors, instead everything is already connected in the way it is supposed to.

The transformation is accomplished by using a list which is used as a stack of available outputs. For example, a `Program` takes one output from the stack (since it has one output), creates a `CoProgram` using this output and puts in on the head of the stack. A `Platform` on the other hand does not take anything from the stack (since it does not have any outputs), and puts a `CoPlatform` on the stack. The other constructors perform similar stack transformations. At the end of the transformation, the head of the stack should contain the entire `CoDiag` structure. A stack is necessary to make sure that a `Compiler` can take two structure from the stack (since it has two outputs). The stack is initiated with an infinite list of `CoNothing`s, to allow for undefined outputs.

Only a few checks are required on the `Diag` structure, the rest of the checking can be done on the `CoDiag` structure. The checks that are necessary are:

- Check whether there are no elements that are not `CoNothing` left on the stack: this means we have defined too many outputs. Fail if this is the case.
- Check whether there are too few outputs defined, i.e. some of the `CoNothing`s have been consumed. Give a warning if this is the case.
- Check that you compile with a compiler.
- Check that you execute on a interpeter or platform.

Typing rules for `CoDiag`

We can run things on `CoNothing`, a `CoPlatform` or a stack of interpreters ending in a `CoNothing` or `CoPlatform`:

$$\frac{m \ : \ \texttt{Ident}}{\texttt{CoNothing} \ : \ \texttt{TRunner}\ m}\ \textsc{Empty runner}$$

$$\frac{m : \texttt{Ident}}{\texttt{CoPlatform}\ m : \texttt{TRunner}\ m}\ \textsc{Platform runner}$$

$$\frac{d \ : \ \texttt{TRunner}\ m \qquad i, l, m \ : \ \texttt{Ident}}{\texttt{CoInterpreter}\ i\ l\ m\ d \ : \ \texttt{TRunner}\ l}\ \textsc{Interpreter runner}$$

We can either run the result of a compiler directly, or we can compile it with another compiler:

$$\frac{d_1 \ : \ \texttt{TRunner}\ l_2 \qquad d_2 \ : \ \texttt{TRunner}\ m \qquad c, l_1, l_2, m \ : \ \texttt{Ident}}{\texttt{CoCompiler}\ c\ l_1\ l_2\ m\ d_1\ d_2 : \texttt{TCompiler}\ l_1}\ \textsc{Compiler stack rule 1}$$

$$\frac{d_1 \ : \ \texttt{TCompiler}\ l_2 \qquad d_2 \ : \ \texttt{TRunner}\ m \qquad c, l_1, l_2, m \ : \ \texttt{Ident}}{\texttt{CoCompiler}\ c\ l_1\ l_2\ m\ d_1\ d_2 : \texttt{TCompiler}\ l_1}\ \textsc{Compiler stack rule 2}$$

We can compile the program on a compiler stack, or run it directly:

$$\frac{d \ : \ \texttt{TCompiler}\ l \qquad p, l \ : \ \texttt{Ident}}{\texttt{CoProgram}\ p\ l\ d \ : \ \texttt{TProgram}}\ \textsc{Program rule 1}$$

$$\frac{d \ : \ \texttt{TRunner}\ l \qquad p, l \ : \ \texttt{Ident}}{\texttt{CoProgram}\ p\ l\ d \ : \ \texttt{TProgram}}\ \textsc{Program rule 2}$$

Implementation of these rules

These rules suggest that, in order to check the type of a `CoDiag`, we need to look at the children and see if those are of the appropriate type. In the implementation however, we look at the parents and look at their type. We also do the checking of the "types" (i.e. is it a compiler, platform, interpreter or program?) separately from the checking whether the names of the platforms/languages match.