T-Diagrams Compiler

João Paulo Pizani Flor   Liewe Thomas van Binsbergen

Saturday, 9th March, 2013

1 Tc-Tdiag, the typechecker

The input of the typechecker is an ATerm representing a tree of diagrams. The inner nodes of this tree are *'executions'* and *'compilations'*. The leaves are the basic diagrams: programs, platforms, interpreters and compilers. Interpreters and platforms can be used for executions, we will call them executors. Programs, interpreters and compilers can be executed and compiled, we will give them the common name programs. A compiler is the only object that can perform a compilation.

The goal of the typechecker is to disallow nonsensical constructions, allowing as many sensible constructions as possible. How many diagrams the system supports is influenced by a number of design choices which will be explained "on the fly" while explaining how the typechecker works.

We have implemented our typechecker using the UUAG Attribute Grammar system.

1.1 Representing types

A typechecker often has type-variables, type-functions and the application of a type-function to a type-variable.

In our type system, applications of type-functions correspond to the inner nodes of the Diagram tree, namely 'executions' and 'compilations'.

In our implementation, language names (e.g. "java") and machine-names (e.g. "amd64-unix") form the type variables. They are simply represented as Strings. However, we need a type that represents a function not having any type and a type that represents all types. They are represented by 'NoType' and 'AnyType' respectively. We need these special types for our implementation of type-functions.

A diagram is such a type-function. They have either both input and output (interpreters and compilers), only input (platform) or only output (program). A type-function with an input expects a certain input, one goal of the typechecker is to compare the expected input with the actual input. Input and output are represented by a single type-variable.

There is only one kind of output, namely the language the object of the diagram is constructed in. This output determines on what type of executor this program can be executed. An input language can be either used for execution or for compilation, which requires us to have two separate kinds of input. Programs have in common that they have an output language. Executors have in common that they expect an input language.

The two rules to use are:

1. The application of a type-function should be valid, meaning that the function should expect input of the right kind. So a compilation can only be successful if the function expects in input language for compilation. In other words: the function has to be a compiler. An execution can only be a successful if the function is an executor. These checks are made at the time of application, at the nodes of the tree.

2. The expected input language and the received input language should match. These checks are made at the type-function level, which are the leaves of the tree. At this point the special 'NoType' and 'AnyType' type-variables are useful.

Design Choice: The choice is whether we allow to execute a program when it is not fully completed yet itself, meaning that it is expecting input. For example whether we allow to execute an interpreter that does not execute anything itself. We have decided that we do allow this, which requires us to give 'AnyType' as input to the first child of an execution. The same decision has been made for the input of a compiler.

Design Choice: We could decide that a type-function only returns its input when its input matches the input it expected. Doing this would make errors 'propagate'. For example, an interpreter for Haskell receiving a Java program would give one error. If we decide it will therefore not give any output, that would mean we would get an additional error when we try to execute it. We have decided that errors DO NOT propagate. The user can benefit more by analyzing and correcting an error in isolation than immersed in a sea of other error messages. However when we have an error in a compilation the error does propagate because the result of the compilation is never created. This is because of the transformation we perform that is explained later.

1.2 Implementation of errors

We have implemented the rules by using synthesized attributes `output` and `error`, together with inherited attribute `input` and local fields `expects` and `ident`.

The ident attribute is a unique identifier for a diagram. It consists of a representation of the constructor, the position in the source code, the identifier used in the source code. In case of compilation or execution also a compilation or execution count, coming from inherited attribute `compnr` and `execnr` respectively.

The two checks mentioned in the last subsection are implemented as follows:

1. We use a haskell datatype called TypeFunc and an attribute 'typefunc' to find out whether the second child of an execution is an executor and whether the second child of a compilation is a compiler. And add an error to the errors we already found in the children.

2. We compare the received input and the expected input and return a singleton list with an error inside if they do not match. For comparing we have

defined an Eq instance for the datatype TypeVar. This instance can not be derived automatically because of the special type-variables 'NoType' and 'AnyType'.

1.3 Transformations

Things become interesting when we compile a program, since a compilation will have a result. The typechecker has to take this result into account, as it can be executed or compiled again. To enable this we have implemented a synthesized attribute that transforms the tree of diagrams. Every compilation will transform into the result of this compilation, but only if the expected input language matches the given input language.

Additional transformations that are possible include transforming an executed interpreter into the input program now in the language of the interpreter. Since this is not required we have not implemented it. Another additional transformation could be the transformation of an executed diagram into the diagram itself. This would be especially useful for an executed compiler.

We need a way to get the input and output language of a compiler when transforming a compilation and the second child (where we expect a compiler) of a compilation could well be an execution! So we can not pattern-match on this second child.

Transforming an execution as mentioned above would be a solution. However this would give the problem that a program executed on a platform could endlessly be executed on the same platform, as its output will go unchanged. To obtain the input and output language of a compiler further down the tree we have used a synthesized attribute called 'langs'.

1.3.1 Compiler result only after compilation

We have tried to implement the following rule:

- A compilation will only yield a result if it is sufficiently executed. Meaning that the compiler used in the compilation need to be run on a platform directly or indirectly trough one or more interpreters.

To do this a transformation needs to propagate upwards. Starting from a compilation and being applied when a platform is used for execution. A difficulty is that this chain of executions might contain an interpreter being compiled first. Hence we get a nested set of delayed transformations.

We were almost able to implement this using a stack of delayed transformations, however due to time constraints we were not able to fully work it out. Since it is also not part of the assignment and since the rule set would be different then the one defined in the assignment, we have decided not to include this rule in our program. The code that was written to try to implement can be found in the doc directory.

2 TDiag2Pict - Transforming a T-Diagram into a LaTeXPicture

The job of the "backend" in the case of this assignment is to generate LaTeXcode containing the graphical rendition of the T-Diagram, inside the `picture` environment. This is achieved by two components, that communicate through the use of ATerms:

tdiag2pict This component transforms a T-Diagram (value of type `Diag` in a value of the `Picture` datatype, which was provided in the assignment package.

pp-picture Pretty-printing of a LaTeXpicture, that means, transforming a value of type `Picture` into a string with all the drawing commands available in the latex picture environment. This component was already provided in the assignment package.

Even though there were some modifications done to the `pp-picture` component, most of the work was focused on the `tdiag2pict` executable. The `pp-picture` module was adapted to output a *complete* LaTeX document, instead of just the contents of the picture environment.

The development work leading to the `tdiag2pict` executable can be subdivided in two main parts: first, we developed a drawing library, capable of generating the kind of geometric constructs we needed (lines, texts, rectangles, triangles, etc.); then we devised recusive rules on how to compose these drawings, i.e, make bigger diagrams from smaller ones. Both of these efforts are explained in more detail below.

2.1 Geometry.hs, the drawing library

In the module CCO/Drawing/Geometry.hs we define the main drawing API, which is then used to define the *semantics* of the `Diag` datatype using the UUAG Attribute Grammar System.

The starting point on which to build upon is the `Object` datatype, already provided in the assignment package. Values of this type correspond directly to LaTeX commands inside the picture environment. By combining this Object with a pair of cartesian coordinates, we get the most basic datatype defined in our drawing library: `PIObject`, i.e, a Position-Independent Object. There are functions to create some basic PIObjects like lines and text boxes as well as functions to move them.

A collection of PIObjects is then defined as a `PIDiagram`, a Position-Independent Diagram. There is a `Monoid` instance defined for `PIDiagram` and values of this type can also be moved.

In the `Geometry` module we implemented several functions to draw geometric constructs appropriate to the needs of the assignment, and some of the most important ones are highlighted below:

rect, penta, tshape, etc. These functions receive dimensions receive dimensions (width and height) as a parameter, and evaluate to drawings of shapes (rectangles, pentagons and t-shapes) that are needed to represent, respectively, interpreters, platforms and compilers.

fitWRect, fitWPenta, etc. These functions take a String as an extra argument and create the corresponding shape so that *the given text fits inside the drawing.*

Having the drawing working, we proceeded to combining diagrams, which is descrbed in more detail in the next subsection.

2.2 TElement.hs and Compose.hs, combining diagrams

The graphical rendition of a T-Diagrams is not only the lines and text labels which are actually drawn, but also a *tag* defining the *type* of the diagram and a set of *pivot points*, which are points in the diagram in which it can be bound to other diagrams, depending on which role it is playing in the combination.

This information is combined with the visible geometric content in the `TElement` datatype, defined as below:

```
data PivotTag
    = PvIE -- Pivot in the case where the diagram acts as Input of Execution
    | PvOE -- When the diagram acts as Output of Execution (platform)
    | PvIC -- When the diagram is acting as Input of Compilation
    | PvOC -- When the diagram is acting as Output of Compilation
    | PvIT -- The diagram EXPECTS an Input for Translation
    | PvOT -- The diagram GENERATES an Output of Translation
    deriving (Eq, Show, Ord, Enum, Bounded)

type Pivot = (PivotTag, Pos)

data TEType = TEInterpreter | TEProgram | TEPlatform | TECompiler | TEComposite
    deriving (Eq, Show, Ord, Enum, Bounded)

data TElement = TElement TEType [Pivot] PIDrawing0
    deriving Eq
```

Further in the TElement.hs module we have functions defining TElements for all of the 4 basic diagrams in the T-Diagrams language: programs, interpreters, platforms and compilers. Then, in the module Compose.hs we have defined the rules for the recursive combinations of TElements among themselves.

There are exactly two such combination rules, and they correspond directly to the compound constructs (inner nodes) in the Diag tree:

drawExecution This function receives two TElements as arguments, the first one being the *source* of the execution (geometrically on top) and the second being the *executor*.

drawCompilation Receives two arguments, *source* and *compiler*, and generates a resulting diagram which contains, besides source and compiler, also the *translated target*. Receives an extra argument containing information about the source, so that this information can be copied to the generated target diagram.

These two functions just mentioned need (each of them) two sets of *rules* to work:

pivot matching the first rule set defines *which pivots to match from the left and right child*. The geometrical "merging" of the two child diagrams is done so that the two chosen pivot points (one from each child) are superimposed in the final coordinate space.

parent pivots the second rule obtains the pivot points for a parent diagram, based on the pivot points of the children.

Below there are the pivot matching rules for both the execution and compilation combinators:

```
-- Matching the pivots of source and executor
executionMatch src exe = (l PvIE src, l PvOE exe)

-- Rule for matching the pivots of source program and compiler
compilationSrcMatch src com = (l PvIC src, l PvIT com)

-- Rule for matching pivots of the source-compiler composite AND the produced target
compilationTrgMatch srcPcom trg = (l PvOT srcPcom, l PvOC trg)
```

And, as an example of a parent-pivots choice rule, we show the rule set defining how to derive the pivots for an execution, given the pivots of both source and executor:

```
executionComb src exe =
    catMaybes
        [ g PvIE exe PvIE  -- the PvIE pivot of the parent is the PvIE of the exe child
        , g PvOE src PvOE
        , g PvIC src PvIC
        , g PvOC exe PvOC
        , g PvIT src PvIT
        , g PvOT src PvOT ]
```

These 2 combination functions (`drawExecution` and `drawCompilation`), together with the aforementioned 4 basic TElement construction functions (`program`, `compiler`, etc.), give us 6 functions, corresponding precisely to each of the 6 constructors of the `Diag` datatype.

Having these 6 functions in hand, the last piece in the puzzle in to define some *semantics* for the `Diag` datatype, using these functions. This task is accomplished by the attribute grammar defined in th file CCO/Drawing/AG/Drawing.ag. In the attribute grammar defined there we have a synthesized attribute called `drawing`, and this attribute in the root is exactly the graphical rendition of the full input.