Lab exercise 1 : T-diagrams

Ferdinand van Walree(3874389) and Matthew Swart(5597250)

November 30, 2015

1 Introduction

In this document we explain the design and implementation of a DSL for generating T-diagrams in Latex.

2 Design

This section gives a short explanation of the pipeline of our 'compiler'. There are four programs that together form the pipeline. Those are ParseTDiag.hs, TypeCDiag.hs, TDiag2Picture.hs and PpPicture.hs. ParseTDiag.hs is responsible for parsing the DSL as defined in the project description. TypeCDiag.hs is our implemented typesystem and typechecks the generated ATerms given as output from ParseTDiag.hs. TypeCDiag simply returns the ATerms it receives from ParseTDiag.hs if the input has been considered well-typed. TDiag2Picture.hs receives it input from TypeCDiag.hs and translates the ATerms into a structure containing latex code for generating T-diagrams. Finally PpPicture.hs is passed ATerms from TDiag2Picture.hs and generates the latex code for generating T-diagrams.

3 Typing-system

We won't be talking about our implementation just yet. First we'll tell you what kind of T-diagram structures we do and do not allow. First of all, all non-sensical diagrams as described in the project documentation are considered ill-typed:

- Executing a platform
- Executing a program, interpreter, or compiler on a program or a compiler
- Executing a program, interpreter, or compiler on a nonmatching platform or interpreter
- Compiling a platform

- Compiling a program, interpreter, or compiler with a program, a platform or an interpreter
- Compiling a program, interpreter, or compiler with a compiler for an incompatible source language

Unfortunately we had to place some more restrictions on the diagrams that we allow. We also consider the following diagrams to be ill-typed:

- Executing a program, interpreter, or compiler on a compiled interpreter
- Compiling an executed program, interpreter or compiler
- Compiling a program, interpreter, or compiler on an executed compiler
- Compiling a compiled program, interpreter or compiler using a compiled compiler
- Compiling a program, interpreter, or compiler using a left recursive compilation
- Compiling a right recursive compilation using a compiler

Now the last two items probably won't make a lot of sense just yet. So let us explain. If we are given the input Compile L with R end, where L and R are diagrams. If L is a Compile, then we say that Compile L with R end is left recursive. likewise if R is a Compile, then we say that it is right recursive. So the third-last diagram: Compiling a compiled program, interpreter or compiler using a compiled compiler, basically says that we do not allow a compile to be both left recursive and right recursive at the same time. The second-last diagram says that we cannot Compile some diagrams using a compilation that contains left recursive compilation. In otherwords, we cannot compile some diagram using a compiled compiler that was compiled using another compiler. The last diagram says that we cannot Compile a right recursive compilation using a compiler. In other words, the diagram that we are trying to compile must not have been compiled with a compiled compiler.

We have a formal specification of our typesystem, which you can find in typesystem.pdf.

4 Implementation

4.1 TypeCDiag

The attribute grammar of TypeCDiag can be found in src:CCO/Diag/Ag/Typing.ag. The generated haskell file can be found in src:/CCO/Diag/Ag.ag

The attribute grammar is used to type check a given Diag. This means that we return the same Diag that we were given. First we'll explain the attribute that is used in the attribute grammar. Afterwards we'll explain the type checking that we do and how we do it.

4.1.1 AG Attributes

We have an attribute for Diag and Diag_. It contains the following synthesized and inherited attributes:

- syn tycons : The datatype TyCons is used to distinguish between basic diag constructs, between composite diad constructs and a group of basic diag constructs.
- inh recurs: The datatype Recurs is used to distinguish between left recursive, right recursive and no recursive compilation.
- ty: The datatype Ty contains the TyCons and all other information of the diag. As not all diags contain an equal amount of information we pack them in Maybes.
- tyErrs: A concatenated list of errors in string format.

4.1.2 Type checking

For each Diag_ we will describe how we pass the values to its attributes. Sometimes it is self-explanatory and in that case we will simply skip it.

Program: The Ty of Program simply contains a Prog of type TyCons and its implementation language packed in a Maybe as the source language of the TyInfo. For Interpreter, Compiler, and platform not a lot is different. Only what information is stored as what.

Interpreter: A interpreter stores the code that it can interpreter as its source language, and its implementation language as its platform language.

Compiler: A compiler stores its source language as its source language, target language as its target language, and its implementation language as its platform language.

Platform: A platform simply stores its platform language as its platform language

Execute: An Execute, just like the Compile, has two subdiagrams: d1 and d2. The ty of execute is the ty of d2. But the tycons is Executed. Furthermore both d1 and d2 are Not_recursive. **tyErrs:** Here the actual type checking happens for an Execute. First of all we concatenate the tyErrs of d1 and d2. Next we'll do some typechecking and also concatenate any possible errors. In execute we use four functions, each doing some kind of type-checking:

- checkRunnable: We pass the tyCons of d1 to this function. Then we check if it is of type Runnable using a function called match. The match function establishes super-typing. That is the TyCons Prog, Interp and Comp are also of type Runnable. In otherwords, if the TyCons of d1 is not Prog, Interp or Comp, then the input is ill-typed and we generate an error. Basically this function checks that we are not executing a platform.
- checkFramework: Similarly to checkRunnable, but now with d2 and we check whether its TyCons is of type Framework. Which Interp and PlatF are. Basically this function checks that we are not executing on a program or compiler.
- checkIfMatches: This function checks for us whether we are executing d1 on a compatible platform d2. To do that we pass the ty's of d1 and d2 to this function, and then basically we compare implementation languages with source languages.
- checkExeOrCompile: We do not allow an Execute to be executed on another execution or compilation. So now we need to use the tycons of d2 and check whether it is equal to Executed or Compiled. If so we generate an error.

Compile: To determine the ty of Compile we have to do a translation of the ty of d1 using the ty of d2. For a Ty with TyCons Prog its source language becomes the target language of the ty of d2. For both Interp and Comp their implementation languages become the target language of the ty of d2. The tycons is simply Compiled. Given that recurs is an inherited attribute, we now define for d1 and d2 their recursion. If d1 is a compile then it is Left_recursive and if d2 is a compile it is Right_recursive, otherwise they are Not_recursive. **tyErrs:** Here the type checking of Compile occurs. First like Execute we'll do a checkRunnable. Next we have to check that d2 is in fact a compiler by using checkComp. Then we also use checkIfMatches to check that the source language of the compiler matches the implementation language of d1. As we do not allow an Execute inside of a Compile we have to check if the tycons of d1 and d2 match Executed. We still need to check for recursion in the Compile. Namely both d1 and d2 cannot both be left and right recursive at the same time. Also there cannot be left recursion in d2 and right recursion in d1.

4.2 TDiag2Picture

The attribute grammar of TDiag2Picture can be found in src:/CCO/Picture/Ag/Translation.ag. The generated haskell file can be found in src:/CCO/Picture/Ag.ag

The attribute grammar is used to convert a given type Diag to a type Picture, which includes all the to be generates latex commands for generating T-diagrams. In this section, we will first explain the attributes that are used in the attribute grammar. Afterwards we'll explain what kind of translation from Diag to Picture we are doing exactly. Of course we won't be going in too much detail, for that we recommend checking out Translation.ag.

4.2.1 AG Attributes

For translating a Diag to a Picture we have defined two attributes both containing synthesized and/or inherited attributes. First we have the Pic attribute, which contains a synthesized attributed named pic of type Picture. Picture is a pre-defined datatype, so we won't be explaining that. Pic is also a datatype that contains a Diag. The Pic attribute has a definition for pic, which simply creates a dataconstructor: Picture using the dimensions and commands found from the Diag Diag_ attribute.

So the second attribute is the Diag Diag_ attribute. It contains quite a few synthesized attributes and inherited attributes:

- inh blen: This is the inherited attribute for the Length of a Block. All diagrams are of at least one block length. Compiler for example has a block length of 3: (3 * blen). We now have set the block length to 50. Now it is not necessary to use blen, but it allows us to have simply one place contain the value of 50, and not that it is spread through our code.
- inh pos: This is the left-top coordinate of a diagram.
- syn cmd: This is the list of Commands that are used to generate T-diagrams in latex
- syn diag: This attribute refers to the Diag itself. We'll be needing it for compilation.
- syn dcons: We have defined a datatype DiagCons, which simply contain short names of the Diagrams excluding the extra information. Another attribute that simply makes our code look a bit better, although not necessary.
- syn w: The width of a diagram
- syn h: The height of a diagram

- syn tlen: This is a total length of a diagram. It is not exactly the same as the width of a diagram.
- syn cjoint: This is the position of diagram d2 in a compile.
- syn ejoint: This is the position of diagram d2 in an execute.
- wdepth: This is the depth of left recursion in a compile
- cdepth: This is the depth of right recursion in a compile
- edepth: This is the depth of a execution tree.

4.2.2 Translation

The actual implementation of our Translation is quite complex given that we have a lot of attributes and dependencies of attributes on each other. So what we'll do is, for every Diag_ we'll explain the values that we are giving to the attributes of that specific Diag_. We won't do that for every attribute as a lot of them are self-explanatory. It is also possible that we explain some stuff for a Diag_ that is also relevant for the other Diag_'s.

Platform: We use a function 'platform' to generate the Commands of a Platform given a position. 'platform' is a sort of template function, such functions exist for all the basic diag constructs. Most values given to attributes here are self-explanatory, except for a few. The attribute tlen can be ignored as it's simply never used. Likewise cjoint and ejoint can also never be used, because a platform can never be compiled or executed. Wdepth and cdepth get, just like the other basic constructs, a value of 0, whereas edepth gets the default value of 1. This is because, when we execute a diag for the first time, that diag will count as a value, but with wdepth,cdepth they do not. Only when you have compiles within compiles do those compiles count.

Program: The width of a program includes its block length and its edges. Furthermore the cjoint is located at its right bottom edge, and the ejoint at its left bottom edge.

Interpreter: There's nothing noteworthy to explain about the interpreter.

Compiler: The total length of a compiler is 2 blocks of length and its wide is 3. This is because when we consider that when a compiler is being compiled a part of the first compiler overlaps with the second compiler.

Execute: Given that execute has two diagrams, to generate the Commands of an execute we simply concatenate the Commands of d1 and d2. Furthermore the position of d2 is simply the ejoint of d1. The height of an execute is a bit more tricky. Namely d1 could easily be an execution tree containing a compilation tree. This is also why we needed edepth, we had to count how tall of a execution tree we had. If it is larger than the possibly containing compilation tree, then we use the executions tree height as the height of the diagram. Given that d2 is always below d1 and cannot be wider, we simply take the max of the two. Although we are quite certain it will always be d1. Finally we increment edepth for each execution.

Compile: This is the most complex Diagram. We'll have to go over each attribute and explain how it gets its value:

- loc.cmpl: So this is a local attribute, which is passed a diag_. This diag_ has the same dataconstructor as d1, but contains different information. It calls a function, which is quite similar to the translate function in TypeC-Diag, except that we also have to recurse through the Diags. When we've recursed through the Diags, we have to pass the target language of d2 (the compiler) to the Diag_ that we are compiling.
- loc.cpos: Now we have to determine the position the compiled diag_ should get. Assuming that d1 could have a multiple number of compilations, we'll be needing its tlen attribute for determining the position. We do know that both the compiled diag_ and d1 should have the same height.
- lhs.cmd: We'll do the same as in attribute, but we add the Commands of the compiled diag_
- loc.d1pos: For determining the position of d1, we need to know whether the tree is left-recursive of right-recursive. If it is left-recursive, we know that d1 should have the pos that was given to compile, otherwise we have to move it to the right. How far to the right depends on how deep the wdepth of d2 is and is calculated by once adding the length of a compiler and then adding the tlen of a compiler for each subsequent wdepth. Finally some diags 'pos' should overlap with its compiler, so we move them to the left. The y-coordinate of pos can still be used, because d1 should still be the heighest diagram.
- d1.pos: We need to reuse the position of d1, which is why we first stored it in a local attribute.
- d2pos: The position of d2 in a left-recursive compilation is simply the cjoint of d1. Otherwise d2 comes before d1, which is why we give it the x-coordinate of 'pos', but the y-coordinate is of the cjoint of d1.
- h: CompileHeight computes the height of the compilation. If the compilation is left recursive we can simply take the height of d1. When that

compilation ends we add the height of the compiler. Also given that a right recursive compilation is simply compilers stacked upon eachother, we can continuously add the height of the compiler. We do have to substract the overlap.

- w: CompileWidth computes the width of the compilation. Again, if we have a right recursive compilation we have compilers stacked upon each other, which means that they overlap. So we remove the total length of the compiler and add its length without overlap. We also add the complete width of d1 as d1 can only be a basic constructs. In the case that the compilation is no longer right-recursive or is left-recursive, we will want to add the width of d2 (a compiler) to the width of d1. We also want to add the width of the compiled d1, but to do that we'll first have to remove the actual length of d1 and then twice add the non-overlapping length of d1.
- lten: Simply the total length of d1 plus the length of a compiler plus one block.
- cjoint and ejoint: The cjoint and ejoint of a compile are the cjoint and cjoint of the compiled diag_
- wdepth and cdepth: We increment the wdepth of d2 and the cdepth of d1.