T-Diagrams: Documentation on the Attribute Grammars

Tim Kuipers Beerend Lauwers Frank Wijmans

March 7, 2012

1 Introduction

This document describes the attribute grammars of our T-Diagram checker and compiler. We chose to have dedicated documentation for the agś because it allows for more information and it is difficult to document the .ag files. These files are pre-compiled to Haskell files. Haddock requires that all documentation is 'hooked' to function definitions, but due to this pre-compilation it is difficult to acquire this high documentation standard.

Therefore, regular Haskell code in the .ag files is documented conform the haddock specification. This documentation can be found in the haddock documentation, under the "documentation" folder. Let's now take a look at the attribute grammars.

2 Attributes

We have two attribute grammars (AGś from this point on) respectively for the two parts of the assignments. Firstly for the checking, which is named `TDiagTypecheck.ag`, and secondly the part for conversion to LaTeX code, named `TDiag2Picture.ag`.

2.1 TDiagTypecheck.ag

The grammar synthesises the type of every possible diagram. We have chosen to split the first part in two attributes, one named Diag and another Diag_. The first synthesises a Ty, short for type. And the second has instead of a Ty a method from a $SourcePos \rightarrow Ty$. This way, type errors can be provided a $SourcePos$, as shown in the line `| Diag lhs.ty = @d @pos`. Only type errors use the source position, correct types throw the argument away. The attributes and the semantics are as follows.

```
attr Diag
syn ty :: { Ty }
sem Diag
| Diag lhs.ty = @d @pos

attr Diag_
syn ty :: { SourcePos → Ty }
sem Diag_
| Program     lhs.ty = const $ ProgramType @l
| Platform    lhs.ty = const $ PlatformType @m
| Interpreter lhs.ty = const $ InterpreterType @l @m
| Compiler    lhs.ty = const $ CompilerType @l1 @l2 @m
| Execute     lhs.ty = genericEvaluate @d1.ty @d2.ty "execute"
| Compile     lhs.ty = genericEvaluate @d1.ty @d2.ty "compile"
```

Diag_ is the most important, where you see how the different parts of the diagram are given a correct type. At the `Compile` and `Execute` constructors you see $genericEvaluate :: Ty \rightarrow Ty \rightarrow$

String → (*SourcePos* → *Ty*) used to give meaning to these special types of the language. This method evaluates two types and will catch nonsensical constructs. The following data type is used for types.

```
data Ty = ProgramType LanguageType
| PlatformType PlatformNameType
| InterpreterType LanguageType PlatformNameType
| CompilerType LanguageType LanguageType PlatformNameType
| TyErr [String]
```

There is a type for every basic block. `Compile` and `Execute`, the two 'action types', are always evaluated to one of these blocks as well. Furthermore, the errors are also a type, in this way we can pass on errors that we find during evaluation of `Execute` and `Compile`. We chose to design it in this way to stat close to the true meaning of the language: the basic blocks are what will be printed, the executions/compilations are not visible parts of the language. Those constructs resemble a function on the simpler types.

2.2 TDiagTypecheck.ag

The second part converts a diagram to a LaTeX-like data structure. The most important detail is that the attributes need to synthesise the positioning. We introduce a data type *Block* that has one constructor, also `Block`:

```
data Block =
 Block {
  ins   :: (Coordinate,Coordinate),
  out   :: (Coordinate,Coordinate),
  rpic  :: (Coordinate -> Commands),
  lang  :: String,
  frame :: (Coordinate,Coordinate)
 }
```

A block has the coordinates of its point where it can connect with others, and another point where it connects to others. `Coordinate` is a tuple of two doubles, so we have two *in*- and two *out*-points. Furthermore it has the actual picture function, which takes a `Coordinate` to build a relatively-positioned list of *Command*s. We also store the language, which is needed when assembling `Compiler` blocks. With compilers we have to generate a output with the same name and new language. Lastly, we store the size of the entire block in the *frame* field.

The AG for this program is as follows.

```
attr Diag Diag_ syn pic :: { Block }
attr Diag Diag_ syn leftmost :: { Diag_ }
attr Diag Diag_ syn lang :: { String }
sem Diag_
| Program           lhs.pic = program2commands @p @l
| Platform          lhs.pic = platform2commands @m
| Interpreter       lhs.pic = interpreter2commands @i @l @m
| Compiler          lhs.pic = compiler2commands @c @l1 @l2 @m
| Execute           lhs.pic = connect V @d1.pic @d2.pic
| Compile           lhs.pic = connect H (connect H @d1.pic @d2.pic) (
   basicDiag2cmndsNtranslateTo (@d2.lang) (@d1.leftmost))
| Program           lhs.leftmost = Program @p @l
| Platform          lhs.leftmost = Platform @m
| Interpreter       lhs.leftmost = Interpreter @i @l @m
| Compiler          lhs.leftmost = Compiler @c @l1 @l2 @m
| Execute           lhs.leftmost = @d1.leftmost
```

```
| Compile     lhs.leftmost = @d1.leftmost
| Compiler    lhs.lang     = @l2
| Execute     lhs.lang     = @d1.lang
```

This attribute grammar has three synthesised attributes. Firstly we have the *pic*, which is the actual picture that is built up. The interesting part is at the `Execute` and `Compile` productions, which both call the function $connect :: Out \rightarrow Block \rightarrow Block \rightarrow Block$. This function takes a direction and two blocks and connects them together into one block. The rest of the functions are just methods with type definition $String \rightarrow Picture$. They create a single, simple block. The parameters are used to put labels in the correct places. Using the *rpic* field of a block, we can transpose blocks accordingly to fit parts together.

The $leftmost$ attribute is the block on the left-hand side of a `Compile`. It is only used there to recreate the same block (with names replaced accordingly) on the right-hand side of a `Compile`.

The *lang* attribute is used in a similar manner, and is used to change the language of the block on the right-hand side of a `Compile`.

To create the Picture out of a Block, you can simply take the *rpic* function and apply (0,0). This would result in the list of *Command*s ready for conversion to LaTeX code.