Compiler Construction

Mini Project

B T-Diagrams

The aim of this mini project is to implement a typed domain-specific language for T-diagrams.

T-diagrams are used to visualise the interactions between programs, platforms, interpreters, and compilers. In this mini project, we implement a system that processes textual descriptions of T-diagrams, checks their internal consistency by means of a type system, and translates them into LaTeX-code for rendering the diagrams as simple graphics.

Architecture

The implementation of the system comprises (at least) four main components:

1. A program `parse-tdiag` that consumes and parses textual specifications of T-diagrams (in a domain-specific language to be defined below) and produces ATerms that describe the structure of the specifications.
2. A program `tc-tdiag` that consumes ATerms as produced by the program `parse-tdiag` and that typechecks the represented diagram specifications, producing either descriptive error messages for ill-typed specifications or else, for well-typed specifications, just the ATrems it consumed.
3. A program `tdiag2picture` that consumes ATerms as produced by the programs `parse-tdiag` and `tc-tdiag` and that translates the represented diagram specifications into LaTeX-code for rendering the diagrams, producing ATerms that describe the structure of the generated LaTeX-code.
4. A program `pp-picture` that consumes ATerms as produced by the program `tdiag2picture` and that produces a pretty printing of the represented LaTeX-code.

Complete implementations of the programs `parse-tdiag` and `pp-picture` are already provided with the project distribution; hence, it remains to implement `tc-tdiag` and `tdiag2picture`.

Both the typechecker and the translation from the domain-specific language into LaTeX have to be implemented as attribute grammars in the UUAG system.

Syntax

To formally define the syntax of our domain-specific language, we introduce sets **Ident** and **Diag** of identifiers and diagrams, respectively:

$$\begin{array}{rcll} C, I, L, M, P & \in & \textbf{Ident} & \text{identifiers} \\ D & \in & \textbf{Diag} & \text{diagrams.} \end{array}$$

Note that the metavariables C, I, M, and P all range over the *same* set of identifiers **Ident**, of which we leave the actual representation abstract. The set of diagrams is given by

$$\begin{array}{rcl} D & ::= & \textbf{program } P \textbf{ in } L \mid \textbf{platform } M \\ & \mid & \textbf{interpreter } I \textbf{ for } L \textbf{ in } M \mid \textbf{compiler } C \textbf{ from } L_1 \textbf{ to } L_2 \textbf{ in } M \\ & \mid & \textbf{execute } D_1 \textbf{ on } D_2 \textbf{ end} \mid \textbf{compile } D_1 \textbf{ with } D_2 \textbf{ end}. \end{array}$$

Semantics

The "meaning" of a diagram is defined in terms of its translation to LaTeX-code. Here, we give an informal description of this translation.

Basic blocks. The constructs

$$\textbf{program} \cdots \textbf{in} \cdots ,$$

$$\textbf{platform} \cdots ,$$

$$\textbf{interpreter} \cdots \textbf{for} \cdots \textbf{in} \cdots ,$$

and

$$\textbf{compiler} \cdots \textbf{from} \cdots \textbf{to} \cdots \textbf{in} \cdots$$

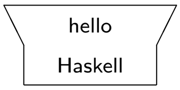
are used to denote so-called *basic blocks.* We give an example for each type of basic block together with its translation into LaTeX-code and the rendering of the associated graphic.

A diagram of the form **program** P **in** L denotes a program P written in some language L. For example, consider

program *hello* **in** *Haskell*

and its LaTeX-translation:

```
\begin{picture}(65,30)
  \put(7.5,0){\line(1,0){50}}
  \put(7.5,0){\line(0,1){15}}
  \put(7.5,15){\line(-1,2){7.5}}
  \put(57.5,15){\line(1,2){7.5}}
  \put(57.5,0){\line(0,1){15}}
  \put(0,30){\line(1,0){65}}
  \put(7.5,15){\makebox(50,15){hello}}
  \put(7.5,0){\makebox(50,15){Haskell}}
\end{picture}
```

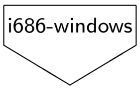


A diagram **platform** M represents the platform referred to by the identifier M. For instance, the diagram

platform *i686-windows*

is mapped to

```
\begin{picture}(50,30)
  \put(0,15){\line(5,-3){25}}
  \put(25,0){\line(5,3){25}}
  \put(0,15){\line(0,1){15}}
  \put(0,30){\line(1,0){50}}
  \put(50,30){\line(0,-1){15}}
  \put(0,15){\makebox(50,15){i686-windows}}
\end{picture}
```

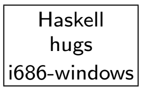


An interpreter I for a language L that itself can be run on a platform of interpreter for the language M is, in our domain-specific language, represented by the diagram **interpreter** I **for** L **in** M. As an example, we have

interpreter *hugs* **for** *Haskell* **in** *i686-windows*

and its LaTeX-rendering

```
\begin{picture}(50,30)
  \put(0,0){\framebox(50,30){}}
  \put(0,20){\makebox(50,10){Haskell}}
  \put(0,10){\makebox(50,10){hugs}}
  \put(0,0){\makebox(50,10){i686-windows}}
\end{picture}
```

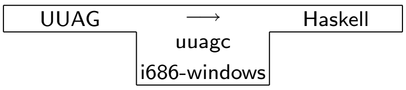


Finally, we use a diagram of the form **compiler** C **for** L_1 **to** L_2 **in** M to represent a compiler C with source languge L_1, target language L_2, and implementation language M. For instance, the diagram

compiler *uuagc* **from** *UUAG* **to** *Haskell* **in** *i686-windows*

is mapped to LaTeX-code as follows:

```
\begin{picture}(150,30)
  \put(50,0){\line(0,1){20}}
  \put(50,20){\line(-1,0){50}}
  \put(0,20){\line(0,1){10}}
  \put(0,30){\line(1,0){150}}
  \put(150,30){\line(0,-1){10}}
  \put(150,20){\line(-1,0){50}}
  \put(100,20){\line(0,-1){20}}
  \put(100,0){\line(-1,0){50}}
  \put(0,20){\makebox(50,10){UUAG}}
  \put(50,20){%
    \makebox(50,10){$\longrightarrow$}}
  \put(100,20){\makebox(50,10){Haskell}}
  \put(50,10){\makebox(50,10){uuagc}}
  \put(50,0){ \makebox(50,10){i686-windows}}
\end{picture}
```



Composite blocks. Diagrams of the forms

execute $\cdots$ **on** $\cdots$ **end**

and

compile $\cdots$ **with** $\cdots$ **end**

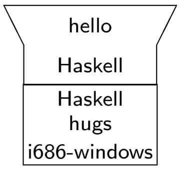
denote *composite blocks.*

A diagram **execute** D_1 **on** D_2 **end** is used to model the execution of the program (interpreter, compiler) represented by the diagram D_1 on the device represented by D_2, typically itself a platform or an interpreter. Such an execution is rendered by drawing the graphical representation of D_1 on top of the representation of D_2. For example, the diagram

execute
  program *hello* **in** *Haskell*
on
  interpreter *hugs* **for** *Haskell* **in** *i686-windows*
    end

is mapped to

```
\begin{picture}(65,60)
  \put(7.5,30){\line(1,0){50}}
  \put(7.5,30){\line(0,1){15}}
  \put(7.5,45){\line(-1,2){7.5}}
  \put(57.5,45){\line(1,2){7.5}}
  \put(57.5,30){\line(0,1){15}}
  \put(0,60){\line(1,0){65}}
  \put(7.5,45){\makebox(50,15){hello}}
  \put(7.5,30){\makebox(50,15){Haskell}}
  \put(7.5,0){\framebox(50,30){}}
  \put(7.5,20){\makebox(50,10){Haskell}}
  \put(7.5,10){\makebox(50,10){hugs}}
  \put(7.5,0){\makebox(50,10){i686-windows}}
\end{picture}
```

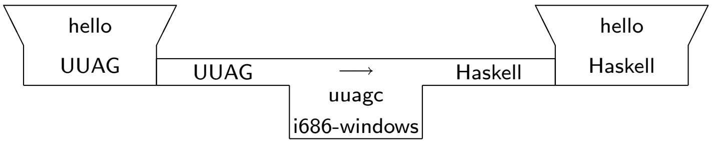


A diagram **compile** D_1 **with** D_2 **end** denotes the compilation of the program (interpreter, compiler) represented by D_1 with the compiler represented by D_2. Compilations are rendered by attaching the graphic for D_1 to the left of the graphic for D_2 *and* by attaching a new graphic, representing the program (interpreter, compiler) produced by the compiler, to the right of the rendering of D_2.

For example, rendering the diagram

compile
  program *hello* **in** *UUAG*
with
  compiler *uuagc* **from** *UUAG* **to** *Haskell* **in** *i686-windows*
   end

results in



```
\begin{picture}(265,45)
  \put(7.5,20){\line(1,0){50}}
  \put(7.5,20){\line(0,1){15}}
  \put(7.5,35){\line(-1,2){7.5}}
  \put(57.5,35){\line(1,2){7.5}}
  \put(57.5,20){\line(0,1){15}}
  \put(0,50){\line(1,0){65}}
  \put(7.5,35){\makebox(50,15){hello}}
  \put(7.5,20){\makebox(50,15){UUAG}}

  \put(107.5,0){\line(0,1){20}}
  \put(107.5,20){\line(-1,0){50}}
```

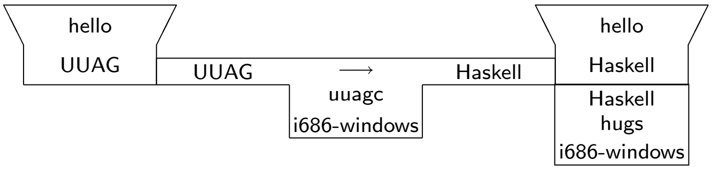
```
  \put(57.5,20){\line(0,1){10}}
  \put(57.5,30){\line(1,0){150}}
  \put(207.5,30){\line(0,-1){10}}
  \put(207.5,20){\line(-1,0){50}}
  \put(157.5,20){\line(0,-1){20}}
  \put(157.5,0){\line(-1,0){50}}
  \put(57.5,20){\makebox(50,10){UUAG}}
  \put(107.5,20){\makebox(50,10){$\longrightarrow$}}
  \put(157.5,20){\makebox(50,10){Haskell}}
  \put(107.5,10){\makebox(50,10){uuagc}}
  \put(107.5,0){\makebox(50,10){i686-windows}}

  \put(207.5,20){\line(1,0){50}}
  \put(207.5,20){\line(0,1){15}}
  \put(207.5,35){\line(-1,2){7.5}}
  \put(257.5,35){\line(1,2){7.5}}
  \put(257.5,20){\line(0,1){15}}
  \put(200,50){\line(1,0){65}}
  \put(207.5,35){\makebox(50,15){hello}}
  \put(207.5,20){\makebox(50,15){Haskell}}
\end{picture}
```

When a compilation-diagram is itself used as a part of a composite diagram, executions and compilations are always to be performed on the synthesized program. For instance, the diagram

```
execute
  compile
    program hello in UUAG
  with
    compiler uuagc from UUAG to Haskell in i686-windows
  end
on
  interpreter hugs for Haskell in i686-windows
   end
```

is rendered as



Type System

The purpose of a type system for T-diagrams is to exclude nonsensical constructions such as executing a Java-program with a Haskell-interpreter or compiling a UUAG-program with a C-compiler. To this end, you should design and implement a type system that excludes nonsensical diagrams, still admitting as many sensible diagrams as possible.

Nonsensical constructions are:

1. executing a platform;

2. executing a program, interpreter, or compiler on a program or a compiler;
3. executing a program, interpreter, or compiler on a nonmatching platform or interpreter;
4. compiling a platform;
5. compiling a program, interpreter, or compiler with a program, a platform, or an interpreter; and
6. compiling a program, interpreter, or compiler with a compiler for an incompatible source langauge.

For the More Ambitious

You may extend the domain-specific language with a facility to bind diagrams to variables, so that you can reuse subdiagrams that occur more than once.

Submitting

The source code of your implementation should be handed in according to the submission instructions on the website of this course.

Submit the source code of your implementation (including both UUAG sources and Haskell sources *not* generated by the UUAG system).

Include in your submission a number of example diagrams (both well-typed and ill-typed).