Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2010/2011

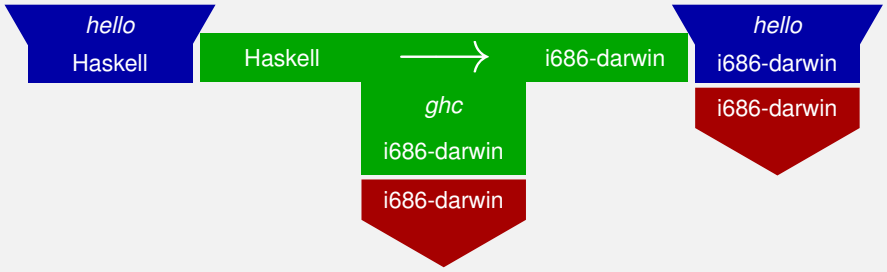
Agenda

Mini project B: T-diagrams

2. Mini project B: T-diagrams

T-diagrams §2

Recall: T-diagrams are a means to visualise the interactions and relationships between programs, platforms, interpreters, and compilers.



☞ T-diagrams should be well-formed: for instance, one cannot execute Java-programs with a Haskell-interpreter.

Drawing T-diagrams

Drawing large, well-formed T-diagrams in LaTeX can be quite involved. A little help is welcome.

Some options:

- Copy-paste.
- Custom LaTeX-macros.
- An embedded domain-specific language for drawing pictures in LaTeX, such as TikZ.
- Custom LaTeX-macros on top of such an EDSL.
- An embedded domain-specific language in Haskell in lieu with lhs2TeX.
- A domain-specific language proper.

A DSL for T-diagrams

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

More specifically, we implement a compilation pipeline that consumes high-level descriptions of T-diagrams and produces LaTeX-code for drawing these diagrams.

Syntax

C, I, L, M, P	$\in$	**Ident**	identifiers
D	$\in$	**Diag**	diagrams

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

☞ The metavariables C, I, L, M, and P range over the *same* set of identifiers.

Semantics: example

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


```
\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}
```

Type system

To enforce the well-formedness of diagrams, we equip our domain-specific language with a type system.

For example, diagrams like

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

should just typecheck.

Type system (cont'd)

Ill-formed diagrams like

```
compile
  program hello in Haskell
with
  compiler javac from Java to JVM in i686-windows
end
```

should be rejected by the type checker.

☞ Hint: the type system should typically involve judgements like

$$\textbf{compiler}\ C\ \textbf{from}\ L_1\ \textbf{to}\ L_2\ \textbf{in}\ M : \mathit{Compiler}\ L_1\ L_2\ M$$

Type system (cont'd)

Devising a set of type rules for just the basic operations like executing a program with an interpreter or compiling a program with a compiler should not be very difficult.

However, things get more involved if we consider that

- programs can also be directly executed on platforms;
- compilers and interpreters are programs as well, so they can be executed and compiled themselves;
- the output program of a compiler can be executed or compiled;
- if the output program of a compiler is an interpreter, it can be used to execute other programs;
- if the output program of a compiler is a compiler, it can be used to compile other programs.

Architecture

As a starting point, we are provided with formal definitions and Haskell implementations of the syntax of the domain-specific language and the syntax of the target language as well as an informal description of the (denotational) semantics for the DSL.

Furthermore, we already have

- a program **parse-tdiag** for parsing T-diagrams and
- a program **pp-picture** for pretty-printing LaTeX-`pictures`.

Then, we are to implement

- a program **tc-tdiag** for typechecking T-diagrams and
- a program **tdiag2picture** for compiling T-diagrams into a `pictures`.

Attribute grammars

The different components will be implemented as attribute grammars.

Using the UU Attribute Grammar Compiler.

Tasks

You will have to:

- Design and formalise a type system for T-diagrams that supports
  - basic operations like executing and compiling a simple program, and
  - more involved diagram compositions.
- Implement the type system.
- Implement a translation from T-diagrams to LaTeX-`pictures`.