Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

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

Edition 2011/2012

Agenda

Case study: An attribute grammar for typechecking arithmetic and boolean expressions

Basic solution

Refinements

6. Case study: An attribute grammar for typechecking arithmetic and boolean expressions

Reference implementations

Available from the course website: some reference implementations of components for interpreters and compilers.

- **parse-arith** and **parse-arithbool**: parsers.
- **pp-arith** and **pp-arithbool**: pretty printers.
- **eval-arith** and **eval-arithbool**: evaluators.
- **tc-arithbool**: type checker.
- **interp-arith** and **interp-arithbool**: complete interpreters.

6.1 Basic solution

Grammar


```
{type Num_ = Int}
data Tm
  | Tm    pos :: {SourcePos} t :: Tm_
data Tm_
  | Num   n :: {Num_}
  | False_
  | True_
  | If    t1 :: Tm t2 :: Tm t3 :: Tm
  | Add   t1 :: Tm t2 :: Tm
  | Mul   t1 :: Tm t2 :: Tm
  | Lt    t1 :: Tm t2 :: Tm
  | Eq    t1 :: Tm t2 :: Tm
  | Gt    t1 :: Tm t2 :: Tm
```

☞ To faciliate error-message generation, every subterm is wrapped in a *Tm*-production that holds a source-file location for the subterm.

Attribution: source positions

We grant each *Tm_* access to its own position by means of an inherited attribute.

Furthermore, through a synthesised attribute, we make the position of each subterm available to its parent.

```
attr Tm_ inh pos :: {SourcePos}
attr Tm  syn pos :: {SourcePos}
```

```
sem Tm
  | Tm t.pos   = @pos
       lhs.pos = @pos
```


Types: syntax

We will not need to perform traversals over the abstract syntax of types, so we can represent types directly in Haskell:

```
{data Ty = Nat | Bool deriving (Eq, Show)}
```

```
{
  instance Tree Ty where
    fromTree Nat  = App "Nat"  []
    fromTree Bool = App "Bool" []
    toTree = parseTree [app "Nat"  (pure Nat )
                       , app "Bool" (pure Bool)
                       ]
}
```

Attribution: typing

We decorate both Tm and $Tm\_$ with an attribute ty in which we will store the type of a term:

$$\begin{array}{l}\mathbf{attr}\ Tm\ Tm\_\\ \quad \mathbf{syn}\ ty :: \{\,Ty\,\}\end{array}$$

Computing these types is straightforward.
That is, if we optimistically assume that typing will succeed:

$$\begin{array}{lll}\mathbf{sem}\ Tm\_ & & \\ \quad \mid Num & \mathbf{lhs}.ty = Nat \\ \quad \mid False\_\ True\_ & \mathbf{lhs}.ty = Bool \\ \quad \mid If & \mathbf{lhs}.ty = @t_2.ty \\ \quad \mid Add\ Mul & \mathbf{lhs}.ty = Nat \\ \quad \mid Lt\ Eq\ Gt & \mathbf{lhs}.ty = Bool\end{array}$$

☞ We assign the ty attributes for $False\_$ and $True\_$ by means of a *single* definition. (And similarly for Add and Mul, and for Lt, Eq, and Gt.)

Attribution: type errors

Of course, we only have to deal with the situation in which a term does not have a valid typing.

Therefore, we associate with each term an attribute $tyErrs$ that is to hold a list of type errors:

$$\begin{array}{l}\mathbf{attr}\ Tm\ Tm\_\\ \quad \mathbf{syn}\ tyErrs\ \mathbf{use}\ \{++\}\ \{[\,]\} :: \{[\,TyErr\,]\}\end{array}$$

☞ A **use**-clause involves two expressions: an infix operator and a default value.
It triggers a so-called *use rule* which, in case of a missing attribute definition, takes precedence over the copy rule.

☞ Relying on the use rule, we do not explicitly define $tyErrs$ for the production Tm of Tm.

Representation of type errors

A $TyErr$-value holds all information that is needed to produce informative type-error messages:

```
{
  data TyErr =
     TyErr SourcePos    -- the location of the error
           String       -- a description of the error
           Ty           -- the expected type
           Ty           -- the acutal type
}
```

$\{\mathbf{instance}\ Printable\ TyErr\ \mathbf{where}\ pp = ppTyErr\}$

☞ $ppTyErr :: TyErr \rightarrow Doc$ produces nicely formatted type-error messages.

A component for type checking

To facilitate wrapping the type checker in a $Component$, we compile the attribute grammar with the appropriate flags and define, within a Haskell module (hence, no curly braces), a function $tyCheck$:

$$\begin{array}{l}tyCheck :: Tm \rightarrow Feedback\ Ty\\ tyCheck\ t = \mathbf{do}\\ \quad \mathbf{let}\ syn = wrap_{Tm}\ (sem_{Tm}\ t)\ Inh_{Tm}\\ \quad messages\ [Error\ (pp\ tyErr) \mid tyErr \leftarrow tyErrs_{Syn|Tm}\ syn]\\ \quad return\ (ty_{Syn|Tm}\ syn)\end{array}$$

☞ From the CCO Library, we use $Error :: Doc \rightarrow Message$ and $messages :: [Message] \rightarrow Feedback\ ()$.

Type checking literals

No type errors can occur in literals.

Hence, relying on the use rule, we do not explicitly define the synthesised attribute $tyErrs$ for the productions Num, $False_{\_}$, and $True_{\_}$.

Type checking conditionals

Type checking a conditional, we first collect the type errors for its three subterms t_1, t_2, and t_3. Then, for further processing, we apply the auxiliary functions $checkTyGuard$ and $checkTyBranches$:

$$
\begin{array}{l}
\textbf{sem } Tm_{\_} \\
\quad \mid If \ \textbf{lhs}.tyErrs \\
\qquad\quad = @t_1.tyErrs +\!\!+ @t_2.tyErrs +\!\!+ @t_3.tyErrs +\!\!+ \\
\qquad\qquad checkTyGuard \ @t_1.pos \ @t_1.ty +\!\!+ \\
\qquad\qquad checkTyBranches \ @t_3.pos \ @t_2.ty \ @t_3.ty
\end{array}
$$

- ☞ $checkTyGuard$ checks that the first subterm is a boolean.
- ☞ $checkTyBranches$ check that the second and third subterm have the same type.
- ☞ Here, the type errors are produced in depth-first order rather than sorted by source position.

Type checking guards

The helper function $checkTyGuard$ checks that the guard of a conditional is of type $Bool$:

$$
\begin{array}{l}
\{ \\
\quad checkTyGuard :: SourcePos \to Ty \to [TyErr] \\
\quad checkTyGuard \ \_ \quad Bool = [\,] \\
\quad checkTyGuard \ pos \ \tau \quad = [TyErr \ pos \ descr \ Bool \ \tau] \\
\qquad \textbf{where} \\
\qquad\quad descr = \texttt{"guard of a conditional"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" should be a boolean"} \\
\}
\end{array}
$$

- ☞ If the check succeeds, no type errors are produced.
- ☞ If the check fails, an appropriate type error is produced.

Type checking branches

The helper function $checkTyBranches$ checks that the branches of a conditional have the same type:

$$
\begin{array}{l}
\{ \\
\quad checkTyBranches :: SourcePos \to Ty \to Ty \to [TyErr] \\
\quad checkTyBranches \ pos \ \tau_{then} \ \tau_{else} \\
\qquad \mid \tau_{then} \equiv \tau_{else} = [\,] \\
\qquad \mid otherwise \quad = [TyErr \ pos \ descr \ \tau_{then} \ \tau_{else}] \\
\qquad \textbf{where} \\
\qquad\quad descr = \texttt{"branches of a conditional"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" should have the same type"} \\
\}
\end{array}
$$

- ☞ If the check succeeds, no type errors are produced.
- ☞ If the check fails, an appropriate type error is produced.

Type checking arithmetic operations

Type checking an addition or a multiplication, we first collect the type errors for its two subterms t_1 and t_2. Then, for further processing, we apply the auxiliary function $checkTyArithOp$:

$$
\begin{array}{l}
\textbf{sem } Tm\_ \\
\quad \mid Add\ Mul\ \textbf{lhs}.tyErrs \\
\qquad\qquad\qquad = @t_1.tyErrs +\!\!+ @t_2.tyErrs +\!\!+ \\
\qquad\qquad\qquad\quad checkTyArithOp\ @t_1.pos\ @t_1.ty +\!\!+ \\
\qquad\qquad\qquad\quad checkTyArithOp\ @t_2.pos\ @t_2.ty
\end{array}
$$

☞ $checkTyArithOp$ checks that the subterms are natural numbers.

Type checking arithmetic operands

The helper function $checkTyArithOp$ checks that the operand of an arithmetic operation is of type Nat:

```
{
  checkTyArithOp :: SourcePos → Ty → [TyErr]
  checkTyArithOp _   Nat = []
  checkTyArithOp pos τ   = [TyErr pos descr Nat τ]
    where
      descr = "operand of an" ++
              " arithmetic operator should" ++
              " be a natural number"
}
```


Type checking relational operations

Type checking a comparison, we first collect the type errors for its two subterms t_1 and t_2. Then, for further processing, we apply the auxiliary function $checkTyRelOp$:

$$
\begin{array}{l}
\textbf{sem } Tm\_ \\
\quad \mid Lt\ Eq\ Gt\ \textbf{lhs}.tyErrs \\
\qquad\qquad\qquad = @t_1.tyErrs +\!\!+ @t_2.tyErrs +\!\!+ \\
\qquad\qquad\qquad\quad checkTyRelOp\ @t_1.pos\ @t_1.ty +\!\!+ \\
\qquad\qquad\qquad\quad checkTyRelOp\ @t_2.pos\ @t_2.ty
\end{array}
$$

☞ $checkTyRelOp$ checks that the subterms are natural numbers.

Type checking relational operands

The helper function $checkTyRelOp$ checks that the operand of a relational operation is of type Nat:

```
{
  checkTyRelOp :: SourcePos → Ty → [TyErr]
  checkTyRelOp _   Nat = []
  checkTyRelOp pos τ   = [TyErr pos descr Nat τ]
    where
      descr =  "operand of a" ++
               " relational operator should" ++
               " be a natural number"
}
```

Example


```
% echo 'if 2 then true else 3 + false fi' | \
> interp-arithbool
line 1:column 4:
  Type error: guard of a conditional should be a boolean.
    expected : Bool
    inferred : Nat

line 1:column 25:
  Type error: operarand of an arithmetic operation should
  be a natural number.
    expected : Bool
    inferred : Nat

line 1:column 21:
  Type error: branches of a conditional should have the
  same type.
    expected : Bool
    inferred : Nat

%
```


6.2 Refinements

Example: bias in typing conditionals


```
% echo 'if false then true else 2 fi + 3' | \
> interp-arithbool
line 1:column 25:
  Type error: branches of a conditional should have
  the same type.
    expected : Bool
    inferred : Nat

line 1:column 1:
  Type error: operand of an arithmetic operator
  should be a natural number.
    expected : Nat
    inferred : Bool

%
```

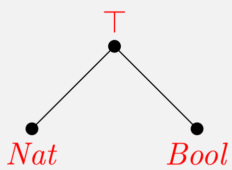
☞ To what extent is the second error message superfluous?

Adding an error type

Idea: if the branches of a conditional are not of the same type, we use neither one of them as the result type of the conditional.

Instead, we use a special type constant $\top$ that denotes a type error.

$\top$ is the greatest element or supremum in a partial order of types:



We have $Nat \sqsubseteq \top$ and $Bool \sqsubseteq \top$.

$(\mathbf{Ty}, \sqsubseteq)$ is a so-called *join-semilattice*: any two types have a least upper bound.

Representing the error type

We extend our representation of types with a constructor for the error type:

$\{\textbf{data}\ Ty = Nat \mid Bool \mid \top\ \textbf{deriving}\ (Eq, Show)\}$

$$
\begin{array}{l}
\{ \\
\quad \textbf{instance}\ Tree\ Ty\ \textbf{where} \\
\qquad fromTree\ Nat \quad = App\ \texttt{"Nat"}\ \ [\,] \\
\qquad fromTree\ Bool = App\ \texttt{"Bool"}\ [\,] \\
\qquad fromTree\ \top \quad\ = App\ \texttt{"Top"}\ \ [\,] \\
\qquad toTree = parseTree\ [app\ \texttt{"Nat"}\ \ (pure\ Nat\ ) \\
\qquad\qquad\qquad\qquad\quad ,\ app\ \texttt{"Bool"}\ (pure\ Bool) \\
\qquad\qquad\qquad\qquad\quad ,\ app\ \texttt{"Top"}\ \ (pure\ \top\ \ ) \\
\qquad\qquad\qquad\qquad\quad] \\
\}
\end{array}
$$

Adapting the type synthesis

Then, we simply change the definition of the synthesised attribute ty for If-productions:

$$
\begin{array}{l}
\textbf{sem}\ Tm\_ \\
\quad \mid If\ \textbf{lhs}.ty = \textbf{if}\ \ \ @t_2.ty \equiv @t_3.ty \\
\qquad\qquad\qquad\quad \textbf{then}\ @t_2.ty \\
\qquad\qquad\qquad\quad \textbf{else}\ \ \top
\end{array}
$$

☞ If the branches have the same type, we produce that the common type; otherwise, we produce the error type.

Least upper bounds

Alternatively, we can define ty for If as

$$
\begin{array}{l}
\textbf{sem}\ Tm\_ \\
\quad \mid If\ \textbf{lhs}.ty = @t_1.ty \sqcup @t_2.ty
\end{array}
$$

Here, $\sqcup$ retrieves the least upper bound or join of two types:

$$
\begin{array}{l}
\{ \\
\quad (\sqcup) :: Ty \to Ty \to Ty \\
\quad \tau_1 \sqcup \tau_2 \\
\qquad \mid \tau_1 \equiv \tau_2 \quad\ = \tau_1 \\
\qquad \mid otherwise = \top \\
\}
\end{array}
$$

☞ A type τ is an upper bound of τ_1 and τ_2 if $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$.

☞ An upper bound τ of τ_1 and τ_2 is the least upper bound of τ_1 and τ_2 if, for any upper bound τ' of τ_1 and τ_2, we have $\tau \sqsubseteq \tau'$.

Example revisited


```
% echo 'if false then true else 2 fi + 3' | \
> interp-arithbool
line 1:column 25:
  Type error: branches of a conditional should have
  the same type.
    expected : Bool
    inferred : Nat

line 1:column 1:
  Type error: operand of an arithmetic operator
  should be a natural number.
    expected : Nat
    inferred : ⊤

%
```

☞ Now the user is confronted with the “internal” representation of type errors.

Nonsyntactic equality for types

Idea: testing a type for equality with $\top$ should always succeed.

This is consistent with our view of $\mathbf{Ty}$ as a partial order: every type is a "subtype" of $\top$.

Hence, we define a nonsyntactic equality test for Ty:

$$
\begin{array}{l}
\{ \\
\quad match :: Ty \to Ty \to Bool \\
\quad match\ \top\ \_\ = True \\
\quad match\ \_\ \top\ = True \\
\quad match\ \tau_1\ \tau_2 = (\tau_1 \equiv \tau_2) \\
\}
\end{array}
$$

- ☞ Two types match if they are same of if one of them is $\top$.
- ☞ Or: two types τ_1 and τ_2 match if either $\tau_1 \sqsubseteq \tau_2$ or $\tau_2 \sqsubseteq \tau_1$.
- ☞ Or: a subset of $\mathbf{Ty}$ is *matching* if it is closed under least upper bounds.

Adapting checkTyGuard

We now adapt the helper function $checkTyGuard$ to use $match$ instead of pattern matching on $Bool$:

$$
\begin{array}{l}
\{ \\
\quad checkTyGuard :: SourcePos \to Ty \to [\,TyErr\,] \\
\quad checkTyGuard\ pos\ \tau \\
\qquad \mid \tau\ \text{`}match\text{`}\ Bool = [\,] \\
\qquad \mid otherwise \qquad = [\,TyErr\ pos\ descr\ Bool\ \tau\,] \\
\qquad \mathbf{where} \\
\qquad\quad descr = \texttt{"guard of a conditional"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" should be a boolean"} \\
\}
\end{array}
$$

Adapting checkTyBranches

Similarly, we adapt $checkTyBranches$ to use $match$ instead of syntactic equality:

$$
\begin{array}{l}
\{ \\
\quad checkTyBranches :: SourcePos \to Ty \to Ty \to [\,TyErr\,] \\
\quad checkTyBranches\ pos\ \tau_{then}\ \tau_{else} \\
\qquad \mid \tau_{then}\ \text{`}match\text{`}\ \tau_{else} = [\,] \\
\qquad \mid otherwise \qquad\quad = [\,TyErr\ pos\ descr\ \tau_{then}\ \tau_{else}\,] \\
\qquad \mathbf{where} \\
\qquad\quad descr = \texttt{"branches of a conditional"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" should have the same type"} \\
\}
\end{array}
$$

Adapting checkTyArithOp

Then we adapt $checkTyArithOp$ to use $match$ instead of pattern matching on Nat:

$$
\begin{array}{l}
\{ \\
\quad checkTyArithOp :: SourcePos \to Ty \to [\,TyErr\,] \\
\quad checkTyArithOp\ pos\ \tau \\
\qquad \mid \tau\ \text{`}match\text{`}\ Nat = [\,] \\
\qquad \mid otherwise \qquad = [\,TyErr\ pos\ descr\ Nat\ \tau\,] \\
\qquad \mathbf{where} \\
\qquad\quad descr = \texttt{"operand of an"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" arithmetic operator should"} +\!\!+ \\
\qquad\qquad\qquad \texttt{" be a natural number"} \\
\}
\end{array}
$$

Adapting checkTyRelOp

Finally, we adapt $checkTyRelOp$ to use $match$ instead of pattern matching on Nat:

```
{
  checkTyRelOp :: SourcePos → Ty → [TyErr]
  checkTyRelOp pos τ
    | τ `match` Nat = []
    | otherwise     = [TyErr pos descr Nat τ]
   where
     descr = "operand of a" ++
             " relational operator should" ++
             " be a natural number"
}
```


Example revisited


```
% echo 'if false then true else 2 fi + 3' | \
> interp-arithbool
line 1:column 25:
  Type error: branches of a conditional should have
  the same type.
    expected : Bool
    inferred : Nat

%
```

☞ Only a single error message is produced.