Type systems
(An introduction)

Ralf Lämmel
Software Language Engineering Team
University of Koblenz-Landau
http://www.softlang.org/
Dependencies:
• Tree-based abstract syntax
• Operational semantics
A type system is a syntax-driven, algorithmic means for rejecting programs that would ‘go wrong’ at runtime.

Abstract syntax of running example

symbol true : → expr ; // The Boolean "true"
symbol false : → expr ; // The Boolean "false"
symbol zero : → expr ; // The natural number zero
symbol succ : expr → expr ; // Successor of a natural number
symbol pred : expr → expr ; // Predecessor of a natural number
symbol iszero : expr → expr ; // Test for a number to be zero
symbol if : expr × expr × expr → expr ; // Conditional

An ill-typed term:

if(zero, false, succ(zero)).

N.B.: The expression language at hand is also referred to as BTL — Basic TAPL Language — where TAPL is a reference to Pierce’s textbook ‘Types and programming languages’.
Well-typed versus ill-typed programs

- **typeOf (Succ Zero)**
  - `Just NatType`

- **typeOf (Succ TRUE)**
  - `Nothing`

The expression **Succ Zero** is of type **NatType**.

The expression **Succ TRUE** is ill-typed.

N.B.: We hint here at how we plan to implement a type system effectively as a type checker. This is comparable to implementing an operational semantics definition as an interpreter.
Type system for BTL's simple expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>true : booltype</td>
<td>t-true</td>
</tr>
<tr>
<td>false : booltype</td>
<td>t-false</td>
</tr>
<tr>
<td>zero : nattype</td>
<td>t-zero</td>
</tr>
<tr>
<td>e : nattype</td>
<td>t-succ</td>
</tr>
<tr>
<td>succ(e) : nattype</td>
<td>t-pred</td>
</tr>
<tr>
<td>pred(e) : nattype</td>
<td>t-iszero</td>
</tr>
<tr>
<td>if(e_0, e_1, e_2) : T</td>
<td>t-if</td>
</tr>
</tbody>
</table>
Typing derivations

N.B.: Subject to some modest constraints on the form of inference rules, these derivation trees can be constructed effectively, thereby ‘typechecking’ programs.
Type safety

• A crucial relationship of ‘agreement’ between
  • (small-step) operational semantics and
  • type system.

• Preservation: If $e : T$ and $e \rightarrow e'$, then $e' : T$.

• Progress: if $e : T$, then $e$ is either
  • a value or
  • there is an $e'$ such $e \rightarrow e'$.

• N.B.: These properties need to be adapted for other languages.
9.3 More type-system examples

--- Types of expressions

data Type = NatType | BoolType

--- Well-typedness of expressions

wellTyped :: Expr → Bool

wellTyped e | Just _ ← typeof e = True
wellTyped e | otherwise = False

--- Types of expressions

typeof :: Expr → Maybe Type

typeof TRUE = Just BoolType
typeof FALSE = Just BoolType
typeof Zero = Just NatType
typeof (Succ e) | Just NatType ← typeof e = Just NatType
typeof (Pred e) | Just NatType ← typeof e = Just NatType
typeof (IsZero e) | Just NatType ← typeof e = Just BoolType
typeof (If e0 e1 e2) |
    Just BoolType ← typeof e0,
    Just t1 ← typeof e1,
    Just t2 ← typeof e2,
    t1 == t2 = Just t1
typeof _ = Nothing

A type checker for BTL’s expressions

N.B.: We implement typing rules in the same way as we previously implemented rules of an operational semantics,
Type system for simple **imperative** programs I/II
(BIPL — Basic Imperative Programming Language)

**Types of statements**

\[
\frac{m \vdash e : T \quad x \notin m}{m \vdash \text{assign}(x, e) : m[x \mapsto T]}
\]

N.B.: Variables are not declared in BIPL. Thus, types of variables are ‘inferred’.

\[
\frac{m \vdash e : T \quad \langle x, T \rangle \in m}{m \vdash \text{assign}(x, e) : m}
\]

\[
\frac{m_0 \vdash s_1 : m_1 \quad m_1 \vdash s_2 : m_2}{m_0 \vdash \text{seq}(s_1, s_2) : m_2}
\]

\[
\frac{m_1 \vdash e : \text{booltype} \quad m_1 \vdash s_1 : m_2 \quad m_1 \vdash s_2 : m_2}{m_1 \vdash \text{if}(e, s_1, s_2) : m_2}
\]

\[
\frac{m \vdash e : \text{booltype} \quad m \vdash s : m}{m \vdash \text{while}(e, s) : m}
\]
Types of expressions

\[ m \vdash \text{intconst}(i) : \text{inttype} \quad \quad \quad [\text{t\textendash}\text{intconst}] \]

\[ m(x) \mapsto T \]
\[ m \vdash \text{var}(x) : T \quad \quad \quad [\text{t\textendash}\text{var}] \]

\[ m \vdash e : T \quad \quad \quad \text{unary}(uo,T) : T' \quad \quad \quad [\text{t\textendash}\text{unary}] \]
\[ m \vdash \text{unary}(uo,e) : T' \]

\[ m \vdash e_1 : T_1 \quad m \vdash e_2 : T_2 \quad \quad \text{binary}(bo,T_1,T_2) : T' \quad \quad \quad [\text{t\textendash}\text{binary}] \]
\[ m \vdash \text{binary}(bo,e_1,e_2) : T' \]

Signatures of operators

\[ \text{unary}(\text{negate},\text{inttype}) : \text{inttype} \quad \quad \quad [\text{t\textendash}\text{negate}] \]

\[ \text{unary}(\text{not},\text{booltype}) : \text{booltype} \quad \quad \quad [\text{t\textendash}\text{not}] \]

\[ \ldots \]
Type system for simple functional programs I/III
(BFPL — Basic Functional Programming Language)

Well-typedness of programs

\[
fs = \langle f_1, \ldots, f_n \rangle \\
fs \vdash f_1 \quad \cdots \quad fs \vdash f_n \\
fs, \emptyset \vdash e : T
\]
\[\vdash \langle fs, e \rangle \]

Well-typedness of functions

\[
fs, [x_1 \mapsto T_1, \ldots, x_n \mapsto T_n] \vdash e : T_0
\]
\[\vdash \langle x, \langle \langle T_1, \ldots, T_n \rangle, T_0 \rangle, \langle x_1, \ldots, x_n \rangle, e \rangle\]
Type system for simple functional programs II/III (BFPL — Basic Functional Programming Language)

**Types of expressions**

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fs, m \vdash \text{intconst}(i) : \text{inttype}$</td>
<td>$\text{t-intconst}$</td>
</tr>
<tr>
<td>$fs, m \vdash \text{boolconst}(b) : \text{booltype}$</td>
<td>$\text{t-boolconst}$</td>
</tr>
</tbody>
</table>

\[
\frac{(x, T) \in m}{fs, m \vdash \text{arg}(x) : T} \quad \text{[t-arg]}
\]

\[
\frac{fs, m \vdash e_0 : \text{booltype} \quad fs, m \vdash e_1 : T \quad fs, m \vdash e_2 : T}{fs, m \vdash \text{if}(e_0, e_1, e_2) : T} \quad \text{[t-if]}
\]

\[
\frac{fs, m \vdash e : T \quad \text{unary}(uo, T) : T'}{fs, m \vdash \text{unary}(uo, e) : T'} \quad \text{[t-unary]}
\]

\[
\frac{fs, m \vdash e_1 : T_1 \quad fs, m \vdash e_2 : T_2}{fs, m \vdash \text{binary}(bo, T_1, T_2) : T'} \quad \text{[t-binary]}
\]

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
BFPL III/III
Well-typedness of BFPL’s function applications

\[
\begin{align*}
fs, m & \vdash e_1 : T_1 \quad \ldots \quad fs, m \vdash e_n : T_n \\
\langle x, \langle \langle T_1, \ldots, T_n \rangle, T_0 \rangle, \langle \langle x_1, \ldots, x_n \rangle, e \rangle \rangle & \in fs \\
\hline
fs, m & \vdash \text{apply}(x, \langle e_1, \ldots, e_n \rangle) : T_0
\end{align*}
\]

[t-apply]

N.B.: Each function application is checked against the signature for the applied function. The body of a function, though, is checked as part of ‘well-typedness of functions’ (I/III).
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See languages BTL, BIPL, and BFPL.
There are Haskell- and Prolog-based type checkers.

The Software Languages Book
http://www.softlang.org/book
The book discusses type systems in more detail.
Other related subjects: lambda calculus.