Type Systems

February 24, 2015
Recall: Type system from Homework 4

### Syntax

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \in \text{Nat}$</td>
<td>$::= \ 0 \mid 1 \mid \ldots$</td>
</tr>
<tr>
<td>$b \in \text{Bool}$</td>
<td>$::= \ \text{tru} \mid \text{fls}$</td>
</tr>
<tr>
<td>$e \in \text{Exp}$</td>
<td>$::= \ \text{litN} n \mid \text{litB} b \mid \neg e \mid \text{cast} e \mid \text{add} e e \mid \text{and} e e \mid \text{equ} e e \mid \text{cond} e e e$</td>
</tr>
</tbody>
</table>

### Type system

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \in \text{Type}$</td>
<td>$::= \ \text{int} \mid \text{bool}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{litN} n$</td>
<td>$\text{int}$</td>
</tr>
<tr>
<td>$\text{litB} b$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>$\neg e$</td>
<td>$T$</td>
</tr>
<tr>
<td>$\text{cast} e$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>$\text{add} e e$</td>
<td>$\text{int}$</td>
</tr>
<tr>
<td>$\text{and} e e$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>$\text{equ} e e$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>$\text{cond} e e e$</td>
<td>$T$</td>
</tr>
</tbody>
</table>
Outline

Introduction
  Concepts and terminology
  The case for static typing

Defining a static type system
  Basic typing relations
  Type soundness

Adding context
Types and type errors

**Type**: a set of syntactic terms that share the same behavior
- `nat, bool, list nat, nat -> bool`
- defines the *interface* for these terms – in what contexts can they appear? what kinds of terms can occur as subexpressions?

**Type error**: occurs when a term cannot be assigned a type
- typically a violation of the type interface between terms
- if not caught/prevented, leads to crash or unpredictable evaluation
A type system detects and prevents/reports type errors.

A language is **type safe** if an implementation can detect all type errors:

- **statically**: by proving the absence of type errors
- **dynamically**: by detecting and reporting type errors at runtime

**Type safe languages**
- Coq, Haskell
- Python, Ruby
- Java

**Unsafe languages**
- C, C++
- PHP, Perl, JavaScript
Implicit type conversions: strong vs. weak typing

Many languages *implicitly convert* between types – is this safe?

- my answer: only if conversion is determined by the type of its context, and *not* by its runtime value

Java

```java
int n = 42;
String s = "Answer: " + n;
```

PHP, Perl

```php
n = "4" + 2;
s = "Answer: " + n
```

**strong typing**: the type of a term is uniquely determined by *its structure and context*

**weak typing**: a term can be assigned multiple types *depending on its runtime value*

In general, must have strong typing to be safe (predictability)
Static vs. dynamic typing

**Static typing**
- types are associated with **syntactic terms**
- type errors are reported at **compile time** (and typically prevent execution)
- type checker **proves** that no type errors will occur at runtime

**Dynamic typing**
- types are associated with **runtime values**
- type errors are reported at **runtime** (e.g. by throwing an exception)
- type checker is **integrated** into the runtime system
Outline

Introduction
  Concepts and terminology
  The case for static typing

Defining a static type system
  Basic typing relations
  Type soundness

Adding context
Benefits of static typing

Usability and comprehension

1. **machine-checked documentation**
   - guaranteed to be correct and consistent with the implementation

2. **better tool support**
   - e.g. code completion, navigation

3. **supports high-level reasoning**
   - by providing named abstractions for shared behavior
Benefits of static typing (continued)

<table>
<thead>
<tr>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. a partial correctness proof – no runtime type errors</td>
</tr>
<tr>
<td>- improves robustness, focus testing on more interesting errors</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. improved code generation</td>
</tr>
<tr>
<td>- can apply type-specific optimizations</td>
</tr>
<tr>
<td>6. type erasure</td>
</tr>
<tr>
<td>- no need for type information or checking at runtime</td>
</tr>
</tbody>
</table>
Potential drawback: static typing is conservative

Q: What is the type of the following expression?
   \[
   \text{if } 3 > 4 \text{ then true else 5}
   \]
A: Static typing: \text{type error}
   Dynamic typing: \text{nat}

Q: What is the type of this one?
   \[
   \text{fun } x:\text{nat} \Rightarrow \text{if } x > 4 \text{ then } x+1 \text{ else false}
   \]
A: Static typing: \text{type error}
   Dynamic typing: \text{???}

Silly examples, but this is a legitimate drawback
- many advanced type features created to “reclaim” expressiveness
Outline

Introduction
   Concepts and terminology
   The case for static typing

Defining a static type system
   Basic typing relations
   Type soundness

Adding context
Static type systems

**Typing relation**: associates a type with a syntactic term

- $e : T \subseteq \text{Exp} \times \text{Type}$  \hspace{1cm} (expression $e$ has type $T$)
- $\Gamma \vdash t : T \subseteq \text{Env} \times \text{Term} \times \text{Type}$  \hspace{1cm} (term $t$ has type $T$ in environment $\Gamma$)

*(Static) type system*: the definition of the typing relation

- typically using inference rules
Static typing is a “static semantics”

**Dynamic semantics (a.k.a. execution semantics)**
- what is the meaning of this program?
- transforms program into value/output (operational)
- describes what program does at runtime

\[ \_ \mapsto \_ \subseteq \text{Term} \times \text{Term} \]
\[ \_ \Downarrow \_ \subseteq \text{Term} \times \text{Value} \]

**Static semantics**
- which programs have meaning?
- classifies/restricts programs based on structure
- programs remain unchanged during static semantics
- describes meaning of program at compile time

\[ \_ : \_ \subseteq \text{Term} \times \text{Type} \]

Typing is a semantics with a more abstract kind of value (types)
Warm-up exercise and midterm review

Syntax

\[ b \in \text{Bool} ::= \text{true} | \text{false} \]
\[ n \in \text{Nat} ::= 0 | 1 | 2 | \ldots \]
\[ e \in \text{Exp} ::= \text{litN} \ n \]
\[ \quad \mid \text{litB} \ b \]
\[ \quad \mid \text{add} \ e \ e \]
\[ \quad \mid \text{cond} \ e \ e \ e \]

1. Define a small-step operational semantics for \( \text{Exp} \). The step relation should have the form:
\[ e \rightarrow e' \subseteq \text{Exp} \times \text{Exp} \]

2. Write the reduction sequences for:

\[ \text{cond true (add (litN 3) (litN 4)) (litN 5)} \]
\[ \text{add (cond (cond (litB false) (litB false) (litB true)) (litN 2) (litN 3)) (litN 4)} \]
Warm-up exercise and midterm review

1. Define a type system for $Exp$. The typing relation should have the form:

$$e : T \subseteq Exp \times Type$$

2. Write a typing derivation for:

- $\text{cond true (add (litN 3) (litN 4)) (litN 5)}$
- $\text{add (cond (cond (litB false) (litB false) (litB true)) (litN 2) (litN 3)) (litN 4)}$
Outline

Introduction
- Concepts and terminology
- The case for static typing

Defining a static type system
- Basic typing relations
- Type soundness

Adding context
Type soundness

How do we know if our type system is a good one?

A type system is **sound** if well-typed programs **don’t lead to runtime type errors**

More specifically, evaluation of a well-typed program should:
- never “get stuck” (progress)
- preserve the type (preservation)

There’s a systematic way to prove this using structural operational semantics!
Type soundness in structural operational semantics

\[ \text{Soundness} = \text{Progress} + \text{Preservation} \]

**Progress** – a well-typed expression is never stuck:

\[ e : T \rightarrow e \in \text{Value} \vee e \mapsto e' \]

**Preservation** – evaluating a well-typed expression yields a well-typed expression:

\[ e : T \land e \mapsto e' \rightarrow e' : T' \text{ typically: } T = T' \]
Outline

Introduction
   Concepts and terminology
   The case for static typing

Defining a static type system
   Basic typing relations
   Type soundness

Adding context
Typing contexts

Often we need to keep track of some information during typing

- types of top-level functions
- types of local variables
- an implicit program stack
- set of declared classes and their methods
- ...

Put this information in the typing context (a.k.a. the environment)

- traditionally named $\Gamma$ or $\Delta$

$$\Gamma \vdash e : T \subseteq Env \times Exp \times Type$$
Exercise

Syntax

\[ \begin{align*}
v & \in \text{Var} \quad ::= \quad a \mid b \mid \ldots \\
e & \in \text{Exp} \quad ::= \quad \text{litN } n \\
& \quad \mid \text{litB } b \\
& \quad \mid \text{add } e \ e \\
& \quad \mid \text{cond } e \ e \ e \\
& \quad \mid \text{LET } v \ e \ e \\
& \quad \mid \text{REF } v
\end{align*} \]

Environment

\[ \begin{align*}
\Gamma & \in \text{Env} \quad ::= \quad \emptyset \\
& \quad \mid v = T ; \Gamma
\end{align*} \]

1. Define a type system for Exp with LET and REF

The typing relation should have the form:

\[ \Gamma \vdash e : T \subseteq \text{Env} \times \text{Exp} \times \text{Type} \]

LET-bound variables are lexically scoped:

Given \( \text{add } (\text{LET } x \ e_1 \ e_2) \ e_3 \)

- the variable \( x \) is bound to the result of \( e_1 \)
- \( x \) can be referred to \( (\text{REF } x) \) within \( e_2 \)
- \( x \) cannot be referred to within \( e_3 \)

Sample expression:

\( \text{(LET } x \ (\text{litB } \text{true}) \text{ (LET } y \ (\text{add } (\text{litN } 3) \ (\text{litN } 5)) \ (\text{cond } (\text{REF } x) \text{ (REF } y) \text{ (LET } x \ (\text{REF } y) \ (\text{REF } x))))))) \)