Types
July 19, 2017
“Now! *That* should clear up a few things around here!”
Outline

Introduction

  Concepts and terminology
    The case for static typing

Implementing a static type system

  Basic typing relations
  Adding context
Types and type errors

**Type**: a set of syntactic terms (ASTs) that share the same behavior

- `Int`, `Bool`, `String`, `Maybe Bool`, `[[Int]]`, `Int->Bool`
- defines the **interface** for these terms — in what contexts can they appear?

**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 a crash or unpredictable evaluation
Type safety

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 errors at runtime

Type safe languages
  • Haskell, SML
  • Python, Ruby
  • Java

Type unsafe languages
  • C, C++
  • PHP, Perl, JavaScript

static
dynamic
mixed
pointers
conversions
Implicit type conversions: strong vs. week typing

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

Only if determined by the *types* and *not* the runtime values!

**Java (safe)**

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

**PHP/Perl (unsafe)**

```perl
n = "4" + 2;
s = "Answer: " + n;
```
Static vs. dynamic typing

Static typing

• types are associated with **syntactic terms**
• type errors are reported at **compile time**
• 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**
• type checker is **integrated** into the runtime system
Outline

Introduction
   Concepts and terminology
      The case for static typing

Implementing a static type system
   Basic typing relations
   Adding context
Benefits of static typing

Usability and comprehension

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

2. **better tool support**
   • code completion, navigation, etc...

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

Correctness

4. **partial proof of correctness** — no runtime type errors
   • improves robustness, focus testing on more interesting errors

Efficiency

5. **improved code generation**
   • can apply type specific optimizations

6. **type erasure**
   • no need for type information or checking at runtime
Drawback of static typing

Conservative

Q: What is the type of the following expression?

\[\text{if } 3 > 4 \text{ then True else 5}\]

A: Static typing: \textit{type error}
Dynamic typing: \textit{Int}

Q: What is the type of the following expression?

\[\lambda x \rightarrow \text{if } x > 4 \text{ then True else } x + 2\]

A: Static typing: \textit{type error}
Dynamic typing: ???
Undecidability of static typing

mayLoop :: Int -> Bool
f x = if mayLoop x then x + 1 else not x

f is **type correct** if \( \text{mayLoop } x \) yields True
f contains a **type error** if \( \text{mayLoop } x \) yields False

Static typing *approximates* by assuming a type error when type correctness cannot be shown — **proven**
Exercise: static vs. dynamic typing

What is the type of the following function under static and dynamic typing?

```haskell
if True then 5 else False
```

Static typing: type error
Dynamic typing: Int

What is the type of the following function under static and dynamic typing?

```haskell
f x = f (not x) ∗ 2
```

Static typing: Bool → Int
Dynamic typing: ???

```
Bool → Int
```
Polymorphism

A value (function, method, etc.) is polymorphic if it can have more than one value.

Different forms of polymorphism can be distinguished based on:

• the relationship between the types
• the implementation of the functions
Forms of polymorphism

Parametric polymorphism
• polymorphic types match a common “type pattern”
• one implementation (e.g. there is only one function)

Ad hoc polymorphism (a.k.a overloading)
• polymorphic types are unrelated
• implementation differs for each type (e.g. different functions are referred to by the same name)

Subtype polymorphism
• types are related by a subtype relation
• one implementation
Outline

Introduction
- Concepts and terminology
- The case for static typing

Implementing a static type system
- Basic typing relations
- Adding context
Static typing is a “static semantics”

**Dynamic semantics** (a.k.a. execution semantics)
- *what is the meaning of this program?*  \[ \text{sem} :: \text{Expr} \rightarrow \text{Val} \]
- relates an AST to a **value**
- describes what program does **at runtime**

**Static semantics**
- *which programs have meaning?*  \[ \text{type0f} :: \text{Expr} \rightarrow \text{Type} \]
- classifies/restricts programs based on structure
- describes what a program does **at compile time**

**Typing is just semantics with a different kind of value!**
Defining a static type system

1. Define the **abstract syntax**, \( E \)
   *the set of abstract syntax trees (ASTs)*

2. Define the structure of **types**, \( T \)
   *another abstract syntax*

3. Define the **typing relation**, \( E : T \)
   *the mapping from ASTs to types*

Then, we can define a dynamic semantics that **assumes** there are no type errors

Example encoding in Haskell:
```haskell
data Exp = ...
data Type = ...
typeOf :: Exp -> Type
```

Implementing a static type system

IntBoolT.hs  PairT.hs
Haskell as a mathematical metalanguage

Math World

Grammars (Languages)  Functions  Sets (Semantic domains)

Data Types

Haskell World

2_syntax  3_semantics

Data Types  Functions  Data Types

Implementing a static type system
Haskell as a mathematical metalanguage

**Math World**

- **Grammars** (Languages)
- **Functions**
- **Sets** (Semantic domains)

**Haskell World**

- **Data Types**
- **Functions**
- **Data Types**

20
Implementing a static type system
Outline

Introduction

- Concepts and terminology
- The case for static typing

Implementing a static type system

- Basic typing relations
  - 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 into the **typing context** (a.k.a. the **environment**)

\[
\text{typeof :: Exp } \rightarrow \text{ Env } \rightarrow \text{ Type}
\]