“Now! *That* should clear up a few things around here!”
4 Types

Types, Type Errors, Type Systems
Why Type Checking?
Static vs. Dynamic Typing
Polymorphism
Parametric Polymorphism
More Motivation ...

Video clip
JavaScript Madness

> "5" - 3
2

> "5" + 3
53

> 5 + "3"
53

> 5 + + "3"
53

> "foo" + + "foo"
fooNaN

> + "foo" + + "foo"
NaN

> + "foo" + + "foo"
NaNfoo
What is a Type (System) ?

• **Type**
  Collection of PL elements that share the same behavior

• **Type System**
  Formal system to characterize the types of PL elements and their interactions
  Can be used to prove the absence of *type errors*

• **Purpose of Type Systems**
  Ensure meaningful interaction of different program parts, i.e. ensure absence of *type errors*
Type Errors

- **Type Error**
  Illegal combination of PL elements
  (typically: applying an operation to a value of the wrong type)

- **Why are type errors bad?**
  Lead to program crashes
  Cause incorrect computations
Why Types are a Good Thing

(1) Types provide *precise documentation* of programs
(2) Types *summarize* a program on an abstract level
(3) Type correctness means *partial correctness* of programs; a type checker delivers partial *correctness proofs*
(4) Type systems can *prevent* runtime *errors* and can save a lot of debugging
(5) Type information can be exploited for *optimization*
Things to Know About Type Systems

(1) Notion of Type Safety
(2) Strong vs. Weak Typing
(3) Static vs. “Dynamic Typing”
(4) Approximation & Undecidability of Static Typing
(5) Type Checking vs. Type Inference
(6) Polymorphism (parametric, subtype, ad hoc)
Example: Expression Language with 2 Types

Expr2.hs
Type Safety

A programming language is called *type safe* if all type errors are detected.

**Type Safe Languages**
- Lisp (*ridiculous type system*)
- Java
- Haskell
- Expr + eval
- Expr + evalDynTC

**Unsafe Languages**
- (type casts, pointers)
- C
- C++
Exercises

(1) Implement an unsafe `eval` function for the language `Expr`
   (a) Use `Int` as the semantic domain
   (b) Map boolean values to 0 and 1

(2) Evaluate unsafe expressions

```haskell
data Expr = N Int
           | Plus Expr Expr
           | Equal Expr Expr
           | Not Expr

Expr2Unsafe.hs
```
Strong vs. Weak Typing

**Strong Typing**
Each value has one precisely determined type

**Weak Typing**
Values can be interpreted in different types
(e.g. “17” can be used as a string or number, or 0 can be used as a number or boolean)

*In practice: Only strongly typed languages are safe (although strong typing does not guarantee safety)*
A Type Checker for the Expression Language

Expr2.hs
TypeCheck.hs
Dynamic vs. Static Typing

**Dynamic Typing**
Types are checked during runtime

**Static Typing**
Types and type errors are found during compile time

<table>
<thead>
<tr>
<th>Statically Typed</th>
<th>Dynamically Typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Lisp</td>
</tr>
<tr>
<td>(Java)</td>
<td></td>
</tr>
<tr>
<td>Expr + evalStatTC</td>
<td>Expr + evalDynTC</td>
</tr>
</tbody>
</table>
Static Typing is Conservative

What is the type of the following expression?

```plaintext
if 3>4 then "hello" else 17
```
Under dynamic typing: Int
Under static typing: type error

How about:

```plaintext
f x = if test x then x+1 else False
```
Under dynamic typing: ?
Under static typing: type error
Exercises

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

\[ f \ x = \text{if not } x \text{ then } x+1 \text{ else } x \]

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

\[ f \ x = f \ (x+1) \ast 2 \]
Piazza Question

Type Checking
Undecidability of Static Typing

\[
\text{test} :: \text{Int} \rightarrow \text{Bool} \\
f \ x = \text{if test x then x+1 else not x}
\]

Since \text{test x} might not terminate, we cannot determinate the value statically because of the undecidability of the halting problem.

\[f\text{ is type correct if test x yields True}
\]
\[f\text{ contains a type error if test x yields False}
\]

Static typing \textit{approximates} by assuming a type error when type correctness cannot be shown.
## Advantages & Disadvantages of Static/Dynamic Typing

<table>
<thead>
<tr>
<th></th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static Typing</strong></td>
<td>prevents type errors</td>
<td>rejects some o.k. programs</td>
</tr>
<tr>
<td></td>
<td>smaller &amp; faster code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>early error detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(saves debugging)</td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Typing</strong></td>
<td>detects type errors</td>
<td>no guarantees</td>
</tr>
<tr>
<td></td>
<td>fewer programming restrictions</td>
<td>slower execution</td>
</tr>
<tr>
<td></td>
<td>faster compilation</td>
<td>released programs may stop</td>
</tr>
<tr>
<td></td>
<td>(&amp; development?)</td>
<td>unexpectedly with type errors</td>
</tr>
</tbody>
</table>
A Type Checker for Arithmetic Language with Pairs

ExprPair.hs
ExprNPair.hs
Haskell as a Mathematical Metalanguage

Math World

Grammars (Languages) → Functions (Semantics) → Sets (Semantic domains)

Data Types

Functions

Data Types

2 Syntax

3 Semantics

Haskell World = Executable Math World
Haskell as a Mathematical Metalanguage

Typing = Static Semantics
Semantics = Dynamic Semantics

Math World

Grammars (Languages) \rightarrow Functions (Typing) \rightarrow Grammars (Types)

Data Types \rightarrow Functions \rightarrow Data Types

Haskell World

2 Syntax
4 Types
Polymorphism

A value (function, method, ...) is *polymorphic* if it has more than one type

Different forms of polymorphism can be distinguished based on:
(a) relationship between types
(b) implementation of functions
Forms of Polymorphism

Parametric Polymorphism
(a) All types match a common “type pattern”
(b) One implementation, i.e., there is only one function

Ad Hoc Polymorphism (aka Overloading)
(a) Types are unrelated
(b) Implementation differs for each type, i.e., different functions are referred to by the same name

Subtype Polymorphism
(a) Types are related by a subtype relation
(b) One implementation (methods can be applied to objects of any subtype)
Parametric Polymorphism

Haskell demo