Types

Outline

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 type errors at runtime

Type safe languages

- Haskell, SML *static*
- Python, Ruby *dynamic*
- Java *mixed*

Unsafe languages

- C, C++ *pointers*
- PHP, Perl, JavaScript *conversions*

Implicit type conversions: strong vs. weak typing

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

Only if it's determined by the **types**, *not* the runtime values!

Java (safe)

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

PHP, Perl (unsafe)

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

Fun diabolical example: http://www.jsfuck.com/
 programming with implicit conversions!

Static vs. dynamic typing

Static typing

- types are associated with **syntactic terms** (ASTs)
- 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

Benefits of static typing

Usability and comprehension

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

Correctness

4. **a partial correctness proof** – 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: static typing is conservative

Q: What is the type of this expression?

```
if 3 > 4 then True else 5
```

A: Static typing: **type error**

Dynamic typing: **`Int`**

Q: What is the type of this one?

```
\x -> if x > 4 then True else x+2
```

A: Static typing: **type error**

Dynamic typing: **???**

Silly examples, but …

- many advanced type features created to “reclaim” expressiveness

Outline

Static typing is a “static semantics”

Dynamic semantics (a.k.a. **execution semantics**)

- *what is the meaning of this program?*
- relates an AST to a **value** (denotational semantics)
- describes meaning of program **at runtime**

```
sem :: Exp -> Val
```

Static semantics

- *which programs have meaning?*
- relates an AST to a **type**
- describes meaning of program **at compile time**

```
typeOf :: Exp -> Type
```

Typing is just a semantics with a different semantic domain

Defining a static type system

Example encoding in Haskell:

1. Define the **abstract syntax**, E
   the set of abstract syntax trees

```
data Exp = ...
```

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

```
data Type = ...
```

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

```
typeOf :: Exp -> Type
```

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

Outline

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**)

```
typeOf :: Exp -> Env -> Type
```