Semantic Analysis – Type Checking

Martin Sulzmann

What are Types good for

Purpose

- Predict/document program behavior: Function expects an integer and yields a Boolean value. Tells us which operations are valid.
- Detect illegal behavior: Add Integer and Booleans.
- Optimization: Boolean values require 1 Byte storage whereas Integer values require at least 2 Bytes.
- ...

Approach

Static (compile-time) versus dynamic (run-time) type checking.

What are Types?

Type Language

$$t \quad ::= \quad \mathsf{Int} \mid \mathsf{Bool} \mid t \rightarrow t$$

Example of a higher-order functional type language.

Static versus Dynamic

- Most PLs check types at compile-time.
- There are type-preserving compilers where the final assembler code is strongly typed.
- Some PLs only check types at run-time.
- Some (scripting) PLs don't care about types at all.

Typing "Rules"

Informal Conditions

- Types of Operands must be compatible.
- if and while must have Boolean conditions.
- ...

Issue

- Good for documentation but too informal.
- Details are missed.
- Ambiguities.
- ...

More Formally

Type Systems

- Formal notation to assign types to programs via a set of typing rules.
- Huge design space (static versus dynamic, strong versus weak, monorphic versus polymorphic, ...)
- We consider specific case:
  - Static typing.
  - Describes the static semantics of program (without actually executing the program).

Type Judgments + Rules

Typing Judgment

```
G |- p : t
```

```
Binding of free variables G = { x1 : t1, ..., xn : tn }
p a program
t it's type
```

Typing Rules

```
G |- e1 : Int    G |- e2 : Int           PREMISE
-------------------------------
G |- e1 + e2 : Int                       CONCLUSION
```

The conclusion follows if we can establish the premise.

Type Checking versus Inference

Full Type Annotations/Checking

In Java, C++ the types of variables and functions must be declared before being used.

Some Type Inference

In Go, C++14 the types of local (automatic) variables can be inferred.

```
// Go example
var y int;
y = 1;
x := y + 3;
```

Type Checking versus Inference (2)

Full Type Inference

In OCaml, Haskell full type inference. What's the type of the following functions?

```
let succ x = x + 1;;

let apply f x = f x;;

let inc x = apply succ x;;
```

Expressive Types

Objective

Accept more programs thanks to expressive/rich types.

Example: Polymorphism

- Subtyping (aka subtype polymorphism)
- Generics (aka parametric polymorphism)

Types for Program Analysis

Objective

Make use of types to (possibly) reject more (illegal) programs.

Example: Types and Effects

- Refine types with effects.
- Effects track "things" that may happen during evaluation.

Example: Type Inference in Haskell/OCaml

Consider

```
let apply f x = f x;;
```

From the program text we derive the following type equations.

```
t_f = t1 -> t2
t_x = t1
```

Hence, we can conclude

```
apply :: (t1 -> t2) -> t1 -> t2
```

where type parameters `t1` and `t2` are generic.

Example: Type Inference in Haskell/OCaml

Consider

```
let succ x = x + 1;;

let apply f x = f x;;

let inc x = apply succ x;;
```

Via type inference (by generating type equations) we can infer that the generic function apply is used in the type context
`(Int -> Int) -> Int -> Int`.

Example: Dimension types for C

Consider the following fragment of a C program.

```
int plus(int, int);

void test() {
  int x = 1;          // in feet
  int y = 2;          // in meters
  int z = plus(1,2);  // physical dimensions do not match !

}
```

Solution: Dimension types. Refine types with dimensions.

```
int<D> plus(int<D>, int<D>);
```

Guarantees that the arguments to plus must have matching dimensions!