CS345H: Programming Languages

Lecture 9: Principles of Typing

Thomas Dillig

Outline

- We will talk about types
- What types compute
- Why types are useful
- Brief survey of types in the real world

Motivation

- When writing programs, everything is great as long as the program works.
- Unfortunately, this is usually not the case
- Programs crash, don't compute what we want them to compute, etc.
- This is a big problem: Arguably, the biggest problem software faces today

Software Correctness

- We would really want to prove that software has the properties we care about
- And in some sense, we seem to have all the ingredients:
  - We have a formal understanding of syntax
  - We have a rigorous mathematic notation to express meaning of programs
  - We even did some proofs in class showing that a small toy program must evaluate to a certain integer
- So what is the problem?

Software Correctness Cont.

- Problem: Rice's theorem. Any non-trivial property about a Turing machine is undecidable
- This means that we can never give an algorithm, that for all programs can decide if this program has an error on some inputs.
- What can we do?
- Give up?

One Approach: Change the Language

- For some properties, we can formulate language rules such that we can detect all errors of this kind before running the program.
- Goal is to remove one source of error from the run-time behavior of programs
- Example: Scoping

Dynamic Scope

- In dynamic scoping, when you use an identifier, it is bound to the most recently defined identifier
- This is dynamic concept; i.e., you in general only know at run-time what variable a name refers to
- Example:
  `fun f with x = x+y in let y = 3 in (f 2)`
- Dynamically scoped languages: LISP, Perl, L
- Dynamic scoping means that you cannot check if identifiers are valid until run-time!

Static Scope

- To avoid this kind of run-time error, we bind every identifier to the closes source code location that defines an identifier with this name
- This means we can check that all identifiers exist at compile time, before running the program
- Example:
  ```
  void foo(int x) {
    int y = x;
    int x = 3;
    int z = x; }
  ```
- Languages with static scoping: C, C++, Java, ML, ...
- Upshot: Can avoid one kind of run-time error by changing the language rules

Dynamic vs. Static Scoping

- Is some cases, changing the rules works well and is the right answer
- Static scoping is such an example.
- While it restricts the kinds of programs you can write, it has another big benefit: Modularity
- With static scope, the behavior of a piece of code is independent of its context, making reuse easier.
- But changing the rules only works in a few cases. What can we do about all the other sources of software errors?

Big Idea

- Big Idea: Just because we cannot prove something about the original program does not mean we cannot prove something about an abstraction of the program.
- Strategy: In addition to the operational semantics, we will also define abstract semantics that will overapproximate the states a program is in
- Example: In L, the operational semantics compute a concrete integer, string or list, while our abstract semantics only compute the if the result is of kind integer, string or list.

Abstraction

- Trick to defining a useful abstraction: Be sure that anything about this abstraction is decidable!
- Consider L and the simple types Int, String, List
- Claim: The abstract value of any expression is decidable
- In other words, we can give an always terminating algorithm for any L program to decide if it evaluates to a String, Int, and List

Abstraction

- Of course, any abstraction will be less precise than the program
- One popular abstraction: types
- Let's assume we have types `Int` and `String`
- Example: `let x = "duck" in x`
- Operational semantics yield concrete value `"duck"`
- Abstract semantics that only differentiate the kind (or type) of the expression yield: `String`

Abstraction

- But we don't just want any abstraction, we need abstractions that overapproximate the result of the concrete program
- Recall the example: `let x = "duck" in x`
- Abstract value `String` overapproximates `"duck"` since `"duck"` is a kind of string
- On the other hand, abstract value `Int` does not overapproximate `"duck"`.

Soundness

- Specifically, we only care about abstract semantics that are sound
- Soundness means that for any program: If we evaluate it under concrete semantics (operational semantics) and our abstract semantics, the abstract value obtained overapproximates the concrete value.

Soundness is Useful

- The reason we only care about sound abstract semantics is the following:
- Theorem: If some abstract semantics are sound and an expression if of abstract value x, then its concrete type y is always part of the abstract value x.
- Why is this useful?
- This means that if a program has no error in the abstract semantics, it is guaranteed not to have an error in the concrete semantics.

Cost of Abstraction

- But using an abstraction comes at a cost:
- What do we know if a a program has an error in the abstract semantics?
- Nothing. We only know that the program may have an error (or not)
- If under some abstract semantics a program has an error, but the program in fact never has this error under concrete semantics, we say this is a false positive
- Finding the right abstractions is key! Abstraction must match properties of interest to be proven.

Types

- In this class, we will focus on one kind of abstraction: types
- This means abstract values are the types in the language
- What is a type? An abstract value representing an (usually) infinite set of abstract values
- Question: For proving what kind of properties are types as abstract values useful?
- Answer: To avoid run-time type errors!

Untyped Languages

- Before we get into types...
- There languages that are untyped
- Example: Assembly language
- `lw $acc $SP-4` will succeed even if `$SP` does not store a pointer
- Untyped $\Rightarrow$ fun memory corruption and undefined semantics if something goes wrong
- We call a language where any type error will be detected (either at run time or compile time) type-safe.
- Important Point: It is impossible to define meaning of non type-safe languages

Dynamically Typed Languages

- Some languages, such as L, are perfectly happy to interpret programs with type errors.
- Example:`4+"duckling"`
- But the type error is still detected at run-time.
- This means that the interpreter or compiler must check the type of every expression and abort if types do not match.
- This strategy is known as dynamic typing.

Static Typing

- Strategy taken by statically typed language:
  - You declare the type on every expression (or the compiler infers it)
  - If types of expressions don't match, compiler refuses to compile your code
- In other words, if for some expression the type the compiler computes includes some value that could cause an error, the compiler rejects it!

Static Typing Cont.

- Big advantage of static typing: Error are detected before running the program!
- Disadvantage: Not every static type error corresponds to a run-time error
- Why? Types are an abstraction! We trade decidability for false positives.
- Consider the following L program:
  `if 0 then 1 else "duck"+4`
- This program does not have a run-time error
- But it has a static type error!

The Type Wars

- Big and still ongoing debate on static vs. dynamic typing today
- Languages with dynamic types: Python, PHP, JavaScript, L
- Languages with static types: Java, OCaml, C, C++
- Advantages of dynamic typing: Rapid prototyping, more correct programs are allowed
- Advantages of static typing: No type errors at run-time

The Type Wars cont.

- Most development uses statically typed languages today.
- But typically, languages include "escape-hatch" for programmers to opt-out of static checking in form of casts
- It is unclear whether this is the best of both worlds or the worst of both worlds!

Type checking vs. Type inference

- We saw earlier that types are just a kind of abstract value
- Two strategies to compute types:
  1. Ask the programmer
  2. Compute types of expressions from the known types of concrete values.
- Most popular languages use strategy (1), known as type checking

Type Checking

- Type checking: The programmer provides some types (typically, every variable) and the compiler complains if some types are inconsistent.
- Languages with type checking: C, C++, Java, ...
- We will (formally) study type checking first.

Type Inference

- In languages with type inference, you don't have to write any types!
- The compiler automatically computes the "best" type of every expression and reports an error if the computed types are not compatible
- Very cool and intriguing idea. We will learn exactly how it works in a few lectures
- There are languages with this feature: ML, Caml, Haskell, Go

Type checking

- When type checking, we first add syntax for types to a language.
- Let's start with the following toy language:

$$
\begin{array}{lcl}
S & \to & \text{integer} \mid \text{string} \mid \text{identifier} \\
 & & \mid S_1 + S_2 \mid S_1 :: S_2 \\
 & & \mid \text{let } id : \tau \; = \; S_1 \text{ in } S_2 \\
\tau & \to & \mathit{Int} \mid \mathit{String}
\end{array}
$$

Operational Semantics

$$
\frac{\text{integer } i}{E \vdash i : i} \qquad \frac{\text{string } s}{E \vdash s : s} \qquad \frac{\text{identifier } id}{E \vdash id : E(id)}
$$

$$
\frac{\begin{array}{c} E \vdash S_1 : i_1 \\ E \vdash S_2 : i_2 \end{array}}{E \vdash S_1 + S_2 : i_1 + i_2} \qquad \frac{\begin{array}{c} E \vdash S_1 : s_1 \\ E \vdash S_2 : s_2 \end{array}}{E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)}
$$

$$
\frac{\begin{array}{l} E \vdash S_1 : e_1 \\ E[x \leftarrow e_1] \vdash S_2 : e_2 \end{array}}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2}
$$

Types

$$
\frac{\text{integer } i}{T \vdash i : \mathit{Int}} \qquad \frac{\text{string } s}{T \vdash s : \mathit{String}} \qquad \frac{\text{identifier } id}{T \vdash id : T(id)}
$$

$$
\frac{\begin{array}{c} T \vdash S_1 : \mathit{Int} \\ T \vdash S_2 : \mathit{Int} \end{array}}{T \vdash S_1 + S_2 : \mathit{Int}} \qquad \frac{\begin{array}{c} T \vdash S_1 : \mathit{String} \\ T \vdash S_2 : \mathit{String} \end{array}}{T \vdash S_1 :: S_2 : \mathit{String}}
$$

$$
\frac{\begin{array}{l} T \vdash S_1 : \tau_1 \\ \tau = \tau_1 \\ T[x \leftarrow \tau] \vdash S_2 : \tau_3 \end{array}}{T \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}
$$