Types for Programs and Proofs

Lecture 1

Ulf Norell

Building reliable software is hard

The scale and complexity of modern systems, the number of people involved in building them, and the range of demands placed on them render it extremely difficult to build software that is even more-or-less correct, much less 100% correct. At the same time, the increasing degree to which information processing is woven into every aspect of society continually amplifies the cost of bugs and insecurities.

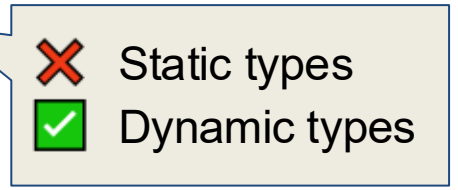
- Software Foundations

Host of techniques for improving software reliability

- **Development processes**
  - extreme programming, agile, scrum
- **Design philosophies**
  - model-view-controller, publish-subscribe
- **Programming languages**
  - object-oriented programming, functional programming
- **Verification techniques**
  - testing, static analysis, formal verification

What are types?

- **A simple view: types as collections of values**
  - e.g. `int` might represent the set of 64-bit integers
  - `v : t` says that a value v is an element of the set represented by `t`
  - or for an expression that it computes to a value in the set
- **Static vs dynamic checking**
  - `if True then 5 else “x” / “y”`



Purpose of (static) types

- **Specification**
  - Documents the programmer's intentions. Checked by the compiler, unlike comments
- **Guide development**
  - IDEs can make suggestions based on types
- **Bug-finding**
  - Catches many common mistakes without having to run the program
- **Optimization**
  - To reserve the minimal amount of memory
  - To use the most appropriate machine instructions

What belongs to type checking?

- **Type checking can prevent**
  - the application of a function to the wrong number of arguments
  - the application of integer functions to strings
  - the use of uninitialized variables
  - null pointer exceptions
  - array indices out of bounds
  - non-terminating programs
  - sorting algorithms that don't sort
- **No language checks all of the above!**

Abstraction, modularity

- **Data encapsulation with abstract types**
  - Guarantees that a program does not rely on implementation details
- **Typed component interfaces in large systems**
  - Always up-to-date documentation
  - Fearless refactoring: when you change the interface the type checker tells you which use sites needs fixing

Polymorphism

- **(or “generics” in Java)**
- **Code reuse**
  - One version works for all types of numbers

```
sum :: Num a => [a] -> a
```

- **Correctness**
  - There’s only one (terminating) function of type

```
forall a. a -> a
```

  - but infinitely many of type

```
Integer -> Integer
```

Example type systems: Rust

Memory safety without garbage collection

```
fn main() {
  let x = "Hello".to_string();
  drop(x);  // free memory
  println!("{}", x);
}
```

```
error[E0382]: use of moved value: `x`
 --> test.rs:6:18
  |
5 |   drop(x);
  |     - value moved here
6 |   println!("{}", x);
  |                  ^ value used here after move
```

https://www.rust-lang.org/

Example type systems: Jif

Java Information Flow: confidentiality and integrity tracked by types

```
public class Booking[principal Owner, principal Operator]
                    authority(Owner) {
  private final String{Owner->*} cardNumber;

  public String{Owner->Operator} getFirstSix{Owner->*}() {
        return cardNumber.substring(0, 6);
        // Error: Return type allows reading by Operator,
    // but field cardNumber does not.
  }
}
```

https://www.cs.cornell.edu/jif/

Example type systems: Liquid Haskell

Enrich Haskell types with arbitrary properties

```
{-@ data IncList a
      = Nil
      | (:<) { hd :: a, tl :: IncList {v : a | hd <= v} }
@-}

insert :: Ord a => a -> IncList a -> IncList a
insert y Nil = y :< Nil
insert y (x :< xs) | y <= x    = y :< x :< xs
                   | otherwise = x :< insert y xs

insertSort :: Ord a => [a] -> IncList a
insertSort []       = Nil
insertSort (x : xs) = insert x (insertSort xs)
```

https://ucsd-progsys.github.io/liquidhaskell-blog/

Example type systems: Coq

Theorem proving using types

```
Fixpoint sumn (n : nat) : nat :=
  match n with
  | 0 => 0
  | (S n') => n + (sumn n')
  end.

Theorem sum_formula : forall n, 2 * (sumn n) = (n + 1) * n.
Proof. intros n. induction n.
        - reflexivity.        (* 0 = 0 base case *)
        - simpl. ring [IHn]. (* induction step *)
Qed.
```

https://coq.inria.fr/

Example type systems: Agda

Mixing programs and proofs. A big part of this course

```
data HasType : Term → Type → Set where
  litRule  : HasType (lit n) int
  appRule  : HasType f (a => b) → HasType e a →
             HasType (app f e) b
  plusRule : HasType plus (int => int => int)

typeCheck : (e : Term) (t : Type) → Maybe (HasType e t)
```

https://wiki.portal.chalmers.se/agda/