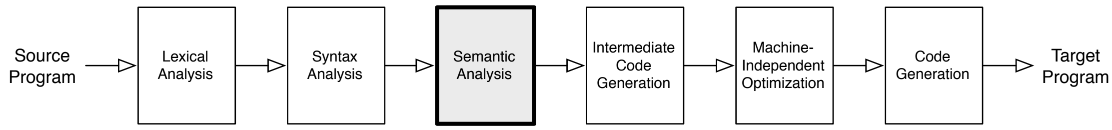
CS 301 Spring 2019
Tutorial Assignment
12 March

Type Polymorphism, Information Flow

1 Plan



This week we discuss type checker design, two forms of type polymorphism, and their intersection: *parametric polymorphism* (generic/parameterized types common in functional languages); and *subtype polymorphism* (foundations of types in object-oriented languages).

We then consider applications of polymorphic type systems for checking *information flow* policies for in security and approximate computing. These topics are supported by research papers. Aim to understand the big ideas and the most important technical details.

2 Readings

Readings on specific topics draw repeatedly on sections (by topic, below) from these sources:

- Two papers on type system foundations. The former introduces formal notations we will use. The latter uses more code examples. Mix and match as they are useful to you.
  - *Type Systems.*
    Luca Cardelli. In *Handbook of Computer Science and Engineering*, CRC Press, 1997.
    `http://lucacardelli.name/Papers/TypeSystems.pdf`
  - *On Understanding Types, Data Abstraction, and Polymorphism.*
    Luca Cardelli and Peter Wegner. *Computing Surveys*, vol. 17 no. 4, December 1985.
    `http://lucacardelli.name/papers/onunderstanding.a4.pdf`
- Programming language references:
  - *The Java Tutorials: Generics (Updated)*: `https://docs.oracle.com/javase/tutorial/java/generics/index.html`
  - Roost Language Specification: `https://cs.wellesley.edu/~cs301/s19/project/spec.pdf`

Specific sections by topic:

- Type systems in general, parametric polymorphism:
  - *Type Systems*: Sections 1 - 3 (review), Section 3 up through Table 15, Sections 4 - 5 up to Table 26
  - *On Understanding Types, Data Abstraction, and Polymorphism*: Sections 1-4
  - *The Java Tutorials: Generics (Updated)*:
    - *Why Use Generics*: `https://docs.oracle.com/javase/tutorial/java/generics/why.html`
    - *Generic Types*: `https://docs.oracle.com/javase/tutorial/java/generics/types.html`
    - *Generic Methods*: `https://docs.oracle.com/javase/tutorial/java/generics/methods.html`
  - Roost Language Specification: Sections 9 - 9.1, 11 - 11.3
- Subtype polymorphism:

- *Type Systems*: Section 6 up through Table 29
- *On Understanding Types, Data Abstraction, and Polymorphism*: Section 6.1
- Roost Language Specification: Sections 9.2, 11.4

- The intersection of parametric and subtype polymorphism:
  - *Type Systems*: Section 6, 9
  - *On Understanding Types, Data Abstraction, and Polymorphism*: Sections 6.2, 8
  - *The Java Tutorials: Generics (Updated).*
    - *Bounded Type Parameters*: `https://docs.oracle.com/javase/tutorial/java/generics/bounded.html`
    - *Generics, Inheritance, and Subtypes*: `https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html`
  - Roost Language Specification: Sections 9.4, 11.5

- Type system applications:
  - *Language-Based Information-Flow Security.* **Sections I - III.**
    Andrei Sabelfeld, Andrew C. Myers. *IEEE Journal on Selected Areas in Communications*, 2003.
    `http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf`
  - *EnerJ: Approximate Data Types for Safe and General Low-Power Computing.* **Sections 1 - 3.1.**
    Adrian Sampson, *et al.*. *PLDI* 2011.
    `https://doi.org/10.1145/1993498.1993518`
    (Off campus: `https://www.cs.cornell.edu/~asampson/media/papers/enerj-pldi2011.pdf`)
    - *EnerJ, The Language of Good-Enough Computing.* (General-audience article.)
      Adrian Sampson, Luis Ceze, Dan Grossman. *IEEE Spectrum.*
      `https://spectrum.ieee.org/computing/software/enerj-the-language-of-goodenough-computing`

3 Exercises

1. **[Roost Compiler]** Design the `typecheck` package for your Roost compiler. This package will support semantic checks outlined in the Roost specification. The main functionality is the type checker, whose job is to take an AST, check that it is well-typed, and decorate each expression (or statement) node with a representation of its type. You will need to add a field to the abstract class for expression nodes (or to a trait that you mix in to relevant AST node classes) to store the AST representation of the type determined by the type checker for the expression.

   Please come to the tutorial meeting with a design detailed enough to discuss the following items:

   (a) Draw the AST for the expression `x + 3 == 7 || a[1] > -x` as it would be represented by your AST data structure. Annotate each node in the AST with the type of the corresponding expression, assuming that `x` : `i64` and `a` : `i64[]`.

   (b) Sketch the implementations of the type checker's code for checking each of the following kinds of AST nodes:

   - a unary expression ($!e$, $-e$, or $\sim e$).
   - an array access
   - a variable access
   - a let expression
   - an assignment

   (c) Other than the decorations described above, what (if any) other changes will you make to the `ast` package to support type checking?

(d) How will your type checker model typing environments? Must it maintain them explicitly or is this information already captured by your symbol tables and AST?

2. **[Parametric Types]** This Roost `map` function takes a function of argument type `i64` and result type `str` and uses it to map all elements of an array of element type `i64` to elements in a new array of element type `str`. Rewrite the `map` function using parametric polymorphism so that it can map source arrays of any element type to result arrays of any element type using a function of the relevant type.

```
fn map(f: (i64) -> str, elems: i64[]) -> str[] {
  let result = new str[elems.length] in {
    let i = 0 in
    while i < elems.length {
      result[i] = f(elems[i]);
      i = i + 1;
    }
    result
  }
}
```

3. Consider the Roost code below. The result expression and result type of the function definition for `f` have been replaced by `expr` and `Type`, respectively.

```
struct S<T, U> {
  field t: T
  field u: U
  method m<U, V>(t: T, u: U, v: V, w: V) -> S<T, U> {
    let s = new S<T, U>() in {
      s.t = self.t;
      s.u = u;
      s
    }
  }
}
fn f<A>(b: S<A, i64>, c: A) -> A {
  if b.u == 0 {
    b.t
  } else {
    c
  }
}
fn g() -> Type {
  let x = new S<i64,i64>() in {
    x.t = 0;
    x.u = 1;
    expr
  }
}
```

For each of the following pairs of expression and type, indicate whether the code is well typed under this replacement. If the replacement is well typed, indicate the type arguments that are inferred for function `f`'s type parameter `A` or method `m`'s type parameters `U` and `V` in this call. If the replacement is not well typed, briefly explain why.

	Replace `expr` with	Replace `Type` with
(a)	`f(x, "hello")`	`str`
(b)	`f(x, 5)`	`i64`
(c)	`x.m(2, 3, 4, 5)`	`S<i64, i64>`
(d)	`x.m(2, 3, 4, "hello")`	`S<i64, i64>`
(e)	`x.m(2, 3, 4, 5)`	`S<T, U>`
(f)	`x.m(2, "hello", false, true)`	`S<i64, str>`
(g)	`x.m(true, "hello", 4, 5)`	`S<bool, str>`
(h)	`f(x.m(2, "hello", false, true), 5)`	`i64`

4. **[Subtypes]** In a type environment Γ such that $\Gamma \vdash \texttt{a} : \texttt{A}$, $\Gamma \vdash \texttt{b} : \texttt{B}$, $\Gamma \vdash \texttt{c} : \mathsf{bool}$, $\Gamma \vdash \texttt{d} : \texttt{D}$, $\texttt{D} <: \texttt{A}$, and $\texttt{B} <: \texttt{A}$, consider the following expressions:

   (a) `if c { 10 } else { 20 }`

   (b) `if c { 10 } else { true }`

   (c) `let x:  A = if c { a } else { b } in { ...  }`

   (d) `let x:  B = if c { a } else { b } in { ...  }`

   (e) `let x:  A = if c { b } else { d } in { ...  }`

   For each of the `if` e_1 e_2 `else` e_3 expressions appearing in this list:

   - Is this if-else expression safe to use (*i.e.*, will its use never lead to run time type errors) in any program in a context satisfying the type environment conditions above?
   - Does the Roost type system allow it?
   - What is the most precise result type that the type system allows for the full if-else expression?

5. Under the same environment as the previous exercise, consider the expression:

   ```
   if c { null } else { a }
   ```

   This expression should type-check with type `A`.

   (a) Define new typing or subtyping rules to allow the Roost type to handle `null` as a value for any structure type, signature type, or the `str` type. It may be helpful to introduce a new type for the `null` value. Your type system does not need to protect against dereferencing `null` at run time, just against using one type of non-`null` value as another incompatible type and against using `null` as a value in a non-reference type (i64, bool, unit).

   (b) Discuss the pros and cons of supporting `null` in the language. Consider alternatives to `null` that you have encountered in other languages. Could they be coded using existing Roost features or with minor changes to the language?

6. Based on the two previous exercises, consider the following set of Roost type definitions:

   ```
   sig I { ... }
   sig J { ... }
   struct K impl I J { ... }
   struct L impl I J { ... }
   ```

   Recall that, under the subtyping extension, Roost structures are similar to Java classes (but without inheritance) and Roost signatures are similar to Java interfaces.

   (a) Assuming `b` : bool and using these definitions, do the typing rules accept the following expression?

   ```
   if b { new K() } else { new L() }
   ```

   If not, fix the typing rules so they do. If so, what type (or types) could the type system give to this expression?

(b) Capturing this case correctly in typing rules is fairly simple, because our system of inference rules supports nondeterministic choice (a.k.a, it can magically guess the right type to use when the type system would otherwise allow more than one type). Implementing this in a type checker is slightly more interesting. Design the logic your type checker can use to determine whether two types (represented as AST nodes) satisfy the subtype relation.

7. Java uses the following *covariant* subtyping rule for array types:

$$\textsc{covariant array subtype}\quad \frac{\tau_1 <: \tau_2}{\tau_1\texttt{[]} <: \tau_2\texttt{[]}}$$

This subtyping rule was introduced in the original Java type system despite being unsound, before the later addition of generic types and methods to the language. It allowed the type system to accept certain standard library methods such as `System.arrayCopy(Object[] source, Object[] dest)`, a general method that copies the elements of any object array to another. Without the covariant array subtyping rules or generics, such methods could not be written in Java. Even though generics make the original justification for this unsound subtyping rule obsolete, it was retained in following versions of Java for backward compatibility.

(a) Demonstrate *why* this rule causes the type system to be unsound (that is, it allows programs that may cause run-time type errors) by writing a short Java program that would cause a run-time type error when executed. Try running your program. If your answer is correct, it really will crash with a run-time type error exception.

(b) The same issues at play in array subtyping are relevant in the interactions of subtyping and parametric polymorphism. Arrays are essentially a special case of a parametric type: consider them as `Array<T>`. Parametric polymorphic types in Java (see *Generics, Inheritance, and Subtypes*) and Roost (plus Roost arrays) are therefore *invariantly* subtyped only:

$$\textsc{invariant subtype}\quad \frac{\tau_1 <: \tau_2}{\tau_1\texttt{<}\tau_3\texttt{>} <: \tau_2\texttt{<}\tau_3\texttt{>}}$$

On the other hand, Scala supports explicit notation of covariant or contravariant subtyping for type parameters in class definitions (*Scala Tour: Variances*: `https://docs.scala-lang.org/tour/variances.html`). To preserve soundness in the presence of variant subtyping of type parameters, the type checker should enforce restrictions on the declarations of fields or methods in classes with explicit variance. Under what circumstances is covariance safe?

8. Consider the following Roost signature definition.

```
sig A { method ma() -> unit }
sig B with A { method mb() -> unit }
sig C with B { method mc() -> unit }
sig D {
  method md(x: B) -> B
}
```

Consider the body of a definition `struct E impl D`.

(a) What methods must `E` define?

(b) Which of the following definitions of method `md` in `E` should be allowed by the Roost type system? Explain why each accepted type is safe and why each rejected type is unsafe.

i. `method md(x: B) -> B { ... }`
ii. `method md(x: A) -> B { ... }`
iii. `method md(x: C) -> B { ... }`

iv. `method md(x: B) -> A { ... }`

v. `method md(x: B) -> C { ... }`

(c) Consider the logic that should fill the two empty rules for type-checking signatures and structures in the ROOST Language Specification, Figure 5. You do not need to write out the rules (they will be rather verbose in this formulation), but do think through the logic and how you might design them.

9. **[Parametric + Subtype Polymorphism]** Consider generic types, subtypes, and their combination:

   (a) What are the key purposes or benefits of each? How do they support polymorphism?

   (b) Using Figure 2 (p. 35) of *On Understanding Types, Data Abstraction, and Polymorphism*, briefly categorize the type systems (and individual features) of Java, Scala, ROOST, and any other statically typed languages you have used, such as ML, TypeScript, or C++. Note especially the *Bounded Type Parameters* section of the Java generics tutorial.

10. **[Information Flow]** *Skim* sections I – III of the *Language-Based Information-Flow Security* paper, which covers the general issue of security and information flow and discusses a number of research issues regarding how to ensure that confidential information does not accidently leak out of a computation. Explain the meaning and importance of the following concepts from the paper:

    (a) *Static information-flow control*

    (b) *Non-interference*

    (c) *Implicit flow*

11. Consider a C-like language of pointers. Expressions and statements have the following syntax:

$$
\begin{array}{rcl}
e & \rightarrow & n \mid x \mid \&x \mid *e \\
s & \rightarrow & x = e \mid x = \texttt{malloc}() \mid *x = e
\end{array}
$$

where n is an integer constant, x is a variable, and `malloc()` allocates an integer or a pointer on the heap (according to the declared type of `x`), and then returns a pointer to that piece of data. The only types are pointers and integers, but pointers can be multi-level pointers. The syntax for types is:

$$\tau \rightarrow \texttt{int} \mid \tau*$$

(a) Write typing rules for all of the expressions and assignment statements. Use judgments of the form $\Gamma \vdash s$ for statements, and judgments of the form $\Gamma \vdash e : \tau$ for expressions.

(b) Now let's extend the types in this language with two type qualifiers `taint` and `trust`, to support *static information flow control* with the type system. *Tainted* data represents data that the program received from external, untrusted sources, such as standard input, a network socket, or a web form input. All of the other data is *trusted.* Some languages provide dynamic or static support for manual tracking of data tainting to, for example, prevent certain forms of security attacks on web programs such as SQL injections.

To model tainting, we extend the set of statements with a `read()` statement that reads an untrusted integer value from an external source:

$$e \rightarrow \ldots \mid \texttt{read}()$$

The syntax for qualified types is:

$$
\begin{array}{rcl}
\tau & \rightarrow & Q\ R \\
R & \rightarrow & \texttt{int} \mid \tau\ * \\
Q & \rightarrow & \texttt{taint} \mid \texttt{trust}
\end{array}
$$

For instance, `trust ((taint int) *)` represents a trusted pointer to a tainted location, and `taint ((taint int) *)` denotes a tainted pointer to a tainted location.

Write appropriate typing rules for expressions n , `x`, `&`x, `*`e, and `read()` for programs with qualified types. Also write a rule for `malloc`.

(c) We want to prohibit the flow of values from untrusted sources into trusted portions of the memory. However, we want to allow flows of values from trusted locations to tainted locations. We can achieve this by defining an appropriate subtyping relation <: between qualified types. First, we define an ordering $\preceq$ between qualifiers:

$$\frac{}{\texttt{trust} \preceq Q} \qquad\qquad \frac{}{Q \preceq Q}$$

We then use the subtyping rule and a subtype-aware assignment rule:

$$\textsc{Subtype}\ \frac{Q \preceq Q'}{QR <: Q'R} \qquad\qquad \textsc{Assign}\ \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash x = e}$$

to enforce the desired control over trusted values. For instance, these rules would make it possible to type-check this code fragment:

```
taint int x;
trust ((trust int) *) y;
y = malloc();
x = *y;
```

Prove that the above program type-checks by showing the proof trees for each of the two assignments.

(d) Write the remaining rule for indirect assignments {*x $= e$}. Illustrate the use of this rule on a small program like the example above.

(e) Consider the following, more general subtyping rules:

$$\textsc{Subtype 1}\ \frac{Q \preceq Q'}{Q\ \texttt{int} <: Q'\ \texttt{int}} \qquad\qquad \textsc{Subtype 2}\ \frac{Q \preceq Q' \qquad T <: T'}{Q\ (T*) <: Q'\ (T'*)}$$

Are these rules sound? If yes, argue why. If not, show a program fragment that type-checks, but yields a type error at run time.

12. Read sections 1 – 3.1 of the *EnerJ* paper. The accompanying general-audience article can help give more context for this work.

(a) What is the main problem the EnerJ type system aims to solve?

(b) How does the type system, with *approximate* and *precise* types, relate to information flow or our trusted/tainted type system?

(c) How do *non-interference* and *implicit flows* manifest in approximate computing? How does the EnerJ type system track and control them?

(d) EnerJ's `context` qualifier supports code that is polymorphic in its containing class's approximation behavior. How is the `context` qualifier similar to or different from type parameters in a parametric polymorphic type system? Can you construct a situation in which additional genericity would improve the expressivity of EnerJ or is the `context` qualifier sufficient for reasonable codes?

(e) EnerJ's *endorsements* are somewhat like casting. Does their treatment in EnerJ seem closer to casting in C (unchecked) or in Java (actual object type checked against cast type at run time to ensure a subtype relationship)?