The Course: Summary

Four (more or less) equal sized units:

- Basics: syntax, semantics, type safety
- Extensions: product/sum/recursive types, continuations/exceptions (abstract machines)
- Advanced features: polymorphism, data abstraction, dynamic typing, references/monads, laziness, subtypes/inheritance/coercion
- Real(istic) languages: Java (Featherweight), garbage collection, ML (Concurrent Min-)

The Moral

There is a scientific theory of programming languages that motivates and is motivated by the practical problems of programming.

- Don’t settle for less!
- It’s a very beautiful theory!
- There’s lots more to be done!

The GUT of PL’s

Type theory is the grand unified theory of programming languages.

- Rigorous framework for specifying dynamic and static semantics.
- Supports reasoning about languages, e.g. type safety.
- Supports reasoning about programs, e.g. time complexity.

PL's as Types

A programming language is “just” a collection of types!

- Modular arithmetic: int.
- Tuples/structures/records: \( \tau_1 \times \tau_2 \).
- Procedures/functions: \( \tau_1 \rightarrow \tau_2 \).
- Variants: \( \tau_1 \uplus \tau_2 \).
- Recursive types: \( \mu \tau.\).
- Generics/polymorphism: \( \forall \tau.\).
- Abstract types: \( \exists \tau.\).
- And so on!
Applicability of Type Theory

Realistic behavioral properties of programming languages:

- Laziness/call-by-need: \( \text{swap}(\tau) \).
- Subtyping: \( \sigma \leq \tau \).
- Coercion: \( \sigma \leq \tau \sim \nu \).
- Inheritance: \( c \leq c' \).
- And so on!

Propositions as Types

Why does type theory work so well as a foundation for programming?

- Wigner: The Unreasonable Effectiveness of Mathematics in Physics. Why is it that mathematics is so effective in modelling physical phenomena?
- Vardi, et. al.: The Unusual Effectiveness of Logic in Computer Science. Why is logic so effective as a tool for computer science?

Utility of Abstract Machines

Realistic implementation properties of programming languages:

- Continuations: C-machine, E-machine.
- Exceptions: C-machine, H-machine.
- Garbage collection: A-machine.
- References, Concurrency and so on!

Propositions as Types

Type theory “works” because of deep connections with logic!

- A type is a specification of behavior.
- A program of a type is a proof of that specification.

As type systems become more expressive, types become as powerful as general mathematics.

Eventually, computer science and mathematics merge into a unified theory of computation (based on type theory).

Propositions as Types

Logic is the science of consequence, or entailment.

\( \Gamma \vdash \phi \) “\( \phi \) is deducible from assumptions \( \Gamma \)”

Here \( \Gamma \) is a set of formulas.

Inference rules govern the logical connectives:

\[
\begin{align*}
\Gamma \vdash \phi, \quad & \Gamma \vdash \psi & \quad \Gamma \vdash \phi \land \psi \\
\Gamma \vdash \phi, \quad & \Gamma \vdash \psi & \quad \Gamma \vdash \phi \land \psi \\
\end{align*}
\]

We may associate proofs with each entailment:

\( \Gamma \vdash \pi : \phi \) “\( \pi \) is a proof of \( \phi \) from assumptions \( \Gamma \)”

But now \( \Gamma \) has the form \( x_1 : \phi_1, \ldots, x_n : \phi_n \); the proof \( \pi \) can refer to these names.

For conjunction we have:

\[
\begin{align*}
\Gamma \vdash \pi_1 : \phi, \quad & \Gamma \vdash \pi_2 : \psi & \quad \Gamma \vdash \pi : \phi \land \psi \\
\Gamma \vdash \pi_1 : \phi, \quad & \Gamma \vdash \pi_2 : \psi & \quad \Gamma \vdash \pi : \phi \land \psi \\
\end{align*}
\]
Propositions as Types

But these are simply the rules for binary tuples!

\[ pfs(\phi_1 \land \phi_2) = pfs(\phi_1) \times pfs(\phi_2) \]

Similar equations govern the other connectives:

\[ pfs(\top) = \text{unit} \]
\[ pfs(\bot) = \text{void} \]
\[ pfs(\phi_1 \lor \phi_2) = pfs(\phi_1) + pfs(\phi_2) \]
\[ pfs(\phi_1 \rightarrow \phi_2) = pfs(\phi_1) \rightarrow pfs(\phi_2) \]
\[ pfs(\neg \phi) = \phi \text{ cont} \ (1) \]

Rules for implication:

\[ \Gamma, \phi \vdash \psi \quad \Gamma, \phi \vdash \psi \quad \Gamma \vdash \phi \]

With proofs:

\[ \Gamma, x : \phi \vdash \pi : \psi \quad \Gamma \vdash \pi_1 : \phi \quad \Gamma \vdash \pi_2 : \phi \]

Proofs of implications are functions!

Where Does This End?

No one knows! It’s the subject of active research!

- **Modal** logic codifies run-time code generation and meta-programming!
- **Linear** logic codifies state change and concurrency.
- **Quantifier** logic codifies reasoning about array bounds.

Much of the research of the the programming languages community is concerned with working out these beautiful connections between logic and computer science.