Type systems

Recall from last week that we had a big issue with our lambda calculus. We were able to construct programs with undefined behavior, i.e. ones that would evaluate to a stuck state, by having free variables in our expressions, like

$$(\lambda\ x\ .\ x)\ y$$

Moreover, we had no way to easily enforce higher-level constraints about our functions. For example, let's say we had a function that would apply an argument twice to a function.

$$\lambda\ f\ .\ \lambda\ x\ .\ f\ x\ x$$

We could accidentally give this a function that only takes one argument[1], e.g.

$$(\lambda\ f\ .\ \lambda\ x\ .\ f\ x\ x)\ (\lambda\ y\ .\ y)$$

Ideally, we could somehow restrict the allowable values for f to the set of functions with two arguments (e.g. $f = \lambda\ x\ .\ \lambda\ y\ .\ x\ y$).

Invariants

The desired properties above are all examples of *invariants*, or program properties that should always hold true. Invariants are things like:

- In the function $\lambda\ n\ .\ 1/n$, n should be a number and $n \neq 0$.
- In my ATM program, customers should not withdraw more money than they have in their account.
- In my TCP implementation, neither party should exchange data until the initial handshake is complete.
- In the driver for my mouse, the output coordinates for the mouse should always be within the bounds of my screen.

There are three main considerations in the design of invariants:

1. **Structure.** What is the "language" of invariants? How can we write down a particular invariant?
2. **Inference.** Which invariants can be inferred from the program, and which need to be provided by the programmer?
3. **Time of check.** When, in the course of a program's execution, is an invariant checked? Before the program is run?

For example, consider the humble `assert` statement. This is usually a built-in function that takes as input a boolean expression from the host language, and raises an error if the expression evaluates to false. In Python:

```
def div(m, n):
    assert(type(m) == int and type(n) == int)
    assert(n != 0)
    return m / n
```

For these kinds of asserts, the language of invariants is the same as the host language, i.e. a Python expression. This is quite powerful! You can do arbitrary computations while checking your invariants. These invariants are never inferred—you have to write the assert statements yourself[2]. And lastly, assert statements are checked at runtime, when the interpreter reaches the assert. Nothing guarantees that an assert is checked before code relying on its invariant is executed (e.g. accidentally dividing and then asserting in the case above).

By contrast, now consider the traditional notion of a "type system" as you know it from today's popular programming languages. For most type systems, the language of invariants is quite restricted—types specify that a variable is a "kind" of thing, e.g. `n` is an `int`, but cannot specify further that `n != 0`. Most stone-age programming languages require the programmer to explicitly provide type annotations (e.g. `int n = 0`), but modern languages increasingly use type inference to deduce types automatically (e.g. `let n = 0`). Lastly, types can be checked either ahead of time ("statically", e.g. C, Java) or during program execution ("dynamically", e.g. Python, Javascript)[3].

Key idea: type systems and runtime assertions derive from the same conceptual framework of enforcing invariants, just with different decisions on when and how to do the checks.

In this course, our focus is going to be on static analysis: what invariants can we describe, infer, and enforce before ever executing the program? And by enforcement, I mean iron law. We don't want our type systems to waffle around with "well, you know, I bet this `n` is going to be an integer, but I'm only like, 75% sure." We expect Robocop type systems that tell us: `I HAVE PROVED TO 100% MATHEMATICAL CERTAINTY THAT IN THE INFINITE METAVERSE OF BOUNDLESS POSSIBILITIES, THIS "n" IS ALWAYS AN INTEGER.`

This is the core impetus behind most modern research in PL theory. Advances in refinement types, dependent types, generalized algebraic data types, module systems, effect systems, traits, concurrency models, and theorem provers have pushed the boundaries of static program analysis. Today, we can prove more complex invariants than ever before. While cutting-edge PL research is mostly beyond the scope of this course, you will be equipped with the necessary fundamentals to continue exploring this space.

Typed lambda calculus

To understand the formal concept of a type system, we're going to extend our lambda calculus from last week (henceforth the "untyped" lambda calculus) with a notion of types (the "simply typed" lambda calculus). Here's the essentials of the language:

$$
\begin{array}{rrll}
\text{Type } \tau ::= & \mathsf{int} & & \text{integer} \\
& \mid & \tau_1 \to \tau_2 & \text{function} \\
\\
\text{Expression } e ::= & x & & \text{variable} \\
& \mid & n & \text{integer} \\
& \mid & e_1 \oplus e_2 & \text{binary operation} \\
& \mid & \lambda\ (x : \tau)\ .\ e & \text{function} \\
& \mid & e_1\ e_2 & \text{application} \\
\\
\text{Binop } \oplus ::= & + \mid - \mid * \mid / &&
\end{array}
$$

First, we introduce a language of types, indicated by the variable tau (τ). A type is either an integer, or a function from an input type τ_1 to an output type τ_2. Then we extend our untyped lambda calculus with the same arithmetic language from the first lecture (numbers and binary operators)[4]. Usage of the language looks similar to before:

$$(\lambda\ (x : \mathsf{int})\ .\ x + 1)\ 2 \mapsto 1 + 2 \mapsto 3$$

$$(\lambda\ (f : \mathsf{int} \to \mathsf{int})\ .\ \lambda\ (x : \mathsf{int})\ .\ (f\ (x + 1)))\ (\lambda\ (y : \mathsf{int})\ .\ y * 2)\ 5 \overset{*}{\mapsto} 12$$

Indeed, our operational semantics are just the lambda calculus plus arithmetic. Zero change from before.

$$\frac{}{n\ \mathsf{val}}\ (\text{D-Int}) \qquad \frac{}{(\lambda\ (x : \tau)\ .\ e)\ \mathsf{val}}\ (\text{D-Lam})$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ (\text{D-App}_1) \qquad \frac{}{(\lambda\ (x : \tau)\ .\ e_1)\ e_2 \mapsto [x \to e_2]\ e_1}\ (\text{D-App}_2)$$

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2}\ (\text{D-Binop}_1) \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'}\ (\text{D-Binop}_2) \qquad \frac{n_3 = n_1 \mathbin{\bar{\oplus}} n_2}{n_1 \oplus n_2 \mapsto n_3}\ (\text{D-Binop}_3)$$

An interpreter for free

A brief aside: the main reason we're using OCaml in this course (as opposed to, say, Haskell or Scala) is that feels quite similar to the typed lambda calculus. In fact, if we change a few keywords, we can use OCaml to execute exactly the language described above. (See the OCaml setup guide to follow along). If we wanted to transcribe the two examples above:

```
$ ocaml
# (fun (x : int) -> x + 1) 2 ;;
- : int = 3
# (fun (f : int -> int) -> fun (x : int) -> f (x + 1)) (fun y -> y * 2) 5 ;;
- : int = 12
```

Of course, OCaml can do much more than this—it has strings, exceptions, if statements, modules, and so on. We'll get there, all in due time. I point this out to show you that by learning the lambda calculus, you are actually learning the principles of real programming languages, not just highfalutin theory. When you go to assignment 2 and start on your first OCaml program, the language will feel more familiar than you may expect!

Type system goals

Before we dive into the type system, it's worth asking the motivational question: what invariants of our language do we want to statically check? One way to answer this is by thinking of edge cases we want to avoid.

- Adding a number and a function: $1 + (\lambda\ (x : \mathsf{int})\ .\ x)$
- Calling a function with the wrong type: $(\lambda\ (x : \mathsf{int} \to \mathsf{int})\ .\ x)\ 0$
- Incorrectly using a function argument: $\lambda\ (x : \mathsf{int})\ .\ (x\ 0)$

This is an important exercise, since it gives us an intuition for where errors might arise. However, even if we had a method for completely eliminating the edge cases we thought of, how can we know we caught *all* the cases? What if we just didn't think of a possible error?

Remember that all of these issues fundamentally boil down to stuck states, or undefined behavior. We specified our operational semantics over "well-defined" programs, but that doesn't prevent us from writing invalid programs. As before, the goal is to take a program and step it to a value. This leads us to a *safety* goal: **if a program is well-defined, it should never enter a stuck state after each step.** If we can formally prove that this safety goal holds for our language, then that means *there are no missing edge cases!*

The goal of a type system, then, is to provide a definition of "well-defined" such that we can prove whether a given program is well-defined *without executing it.* Formally, we need a new judgment (binary relation) "e has type τ", written as $e : \tau$. In the language above, it should be the case that $(1 + 1) : \mathsf{int}$ and $(\lambda\ (x : \mathsf{int})\ .\ x + 1) : \mathsf{int} \to \mathsf{int}$. To say an expression has a type is to say it is "well-defined" (or "well-typed").

Lastly, we want to prove the "type safety" of our language with two theorems:

1. **Progress:** if $e : \tau$ then either $e\ \mathsf{val}$ or there exists e' such that $e \mapsto e'$.

2. **Preservation:** if $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Intuitively, progress says: if an expression is well-typed and is not a value, then we should be able to step the expression (it is not in a stuck state). However, this isn't enough to prove our safety goal, since we also need preservation: if an expression is well-typed, when it steps, its type is preserved. For example, if we have an expression of integer type, it shouldn't turn into a function after being stepped.

Static semantics

In this conceptual framework of type systems, the first thing we need to do is define how we determine the type of an expression. In our grammar, we defined a *type language*, but now we need a *type semantics* (or "static semantics"). First, we'll define the judgments for numbers:

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ (\text{T-Int}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}}\ (\text{T-Binop})$$

As you can see, these are defined quite similarly to how we defined our operational semantics (or "dynamic semantics"). Each rule defines a different way to determine whether a particular expression has a particular type. Just like n val, the T-Int rule of $n : \mathsf{int}$ is axiomatic—a numeric constant has type int under all conditions. T-Binop says: if the two subexpressions are both integers, then the binary operation on those subexpressions is also an integer. From these two rules, we can construct a proof that $(1 + 2 - 3) : \mathsf{int}$:

$$\dfrac{\dfrac{}{\varnothing \vdash 1 : \mathsf{int}}\ (\text{T-Int}) \quad \dfrac{\dfrac{}{\varnothing \vdash 2 : \mathsf{int}}\ (\text{T-Int}) \quad \dfrac{}{\varnothing \vdash 3 : \mathsf{int}}\ (\text{T-Int})}{\varnothing \vdash (2 - 3) : \mathsf{int}}\ (\text{T-Binop})}{\varnothing \vdash (1 + 2 - 3) : \mathsf{int}}\ (\text{T-Binop})$$

Ok, but what is this "$\varnothing \vdash$" business? Or "Γ"? It's the same core idea as what you did for dynamic scoping on Assignment 1. To typecheck expressions with variables, we need to introduce a "typing context" that maps variables to their types. Intuitively, when typechecking $\lambda\ (x : \mathsf{int})\ .\ e$, we want to remember that any usage of x in e should assume $x : \mathsf{int}$. Formally, we write this as:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\text{T-Var}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda\ (x : \tau_1)\ .\ e) : \tau_1 \to \tau_2}\ (\text{T-Lam}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}\ (\text{T-App})$$

Γ represents our type context. (You could also think about it as a "proof context", since our type-checker is basically a theorem prover that's formally verifying the type of an expression.) We can add mappings to our context[5], indicated by $\Gamma, x : \tau$, and we can look up a mapping with $\Gamma(x) = \tau$. Lastly, the notation $\Gamma \vdash e : \tau$ means that "given the proof context Γ, it is provable that $e : \tau$."

Let's read through the rules. T-Var says that if our context says $\Gamma(x) = \tau$ then x has type τ. T-Lam is the most complex: it says that to type-check a function, we want to type-check the body of the function e *assuming* that $x : \tau_1$, where τ_1 is the type provided in the program syntax. Then, assuming our body typechecks to another type τ_2, this becomes the return type of the function, so its entire type is $\tau_1 \to \tau_2$.

> Note a subtlety here: τ_1 is *given* to us from the program, while we have to *compute* τ_2. Our typed lambda calculus mixes types that are *explicitly* annotated and *implicitly* inferred.

Lastly, the T-App rule says: when calling a function, the function expression e_1 should be a function $\tau_1 \to \tau_2$, and the argument expression e_2 should be of the appropriate argument type τ_1. Then, the result of applying the function is the result of the function, type τ_2.

As an example of these rules, here is the proof that $((\lambda\ (x : \mathsf{int})\ .\ x + 1)\ 2) : \mathsf{int}$.

$$\dfrac{\dfrac{\dfrac{\{x : \mathsf{int}\}(x) = \mathsf{int}}{\{x : \mathsf{int}\} \vdash x : \mathsf{int}}\,(\text{T-Var}) \quad \dfrac{}{\{x : \mathsf{int}\} \vdash 1 : \mathsf{int}}\,(\text{T-Int})}{\{x : \mathsf{int}\} \vdash x + 1 : \mathsf{int}}\,(\text{T-Binop})}{\varnothing \vdash (\lambda\ (x : \mathsf{int})\ .\ x + 1) : \mathsf{int} \to \mathsf{int}}\,(\text{T-Lam}) \quad \dfrac{}{\varnothing \vdash 2 : \mathsf{int}}\,(\text{T-Int})}{\varnothing \vdash (\lambda\ (x : \mathsf{int})\ .\ x + 1)\ 2 : \mathsf{int}}\,(\text{T-App})$$

Metatheory

At this point, you should understand the mechanics of our type system: how we define our typing rules, and how they can be used to construct proofs about the types of expressions. But it's not sufficient just to *have* a type system, we need a *good* type system! Remember, we want to demonstrate that if a program is well-typed, then it will never enter a stuck state. To do that, we have to prove the progress and preservation theorems, i.e. verify that our language is actually "type safe." This is an example of a *metatheoretical* property of our programming language. We use the type system to prove that expressions has certain types, and on the next level up we prove that our type system (or proof system) has certain properties.

Structural induction

Before we actually *do* the proof, we need to talk about *how* to do proofs about programming languages. From your discrete mathematics course, you're probably familiar with "mathematical" induction (see CS 103 Mathematical Induction for a refresher) where induction always occurs on the natural numbers. If you want to prove a proposition $P(n)$ for all $n \in \mathbb{N}$, then an inductive proof will show $P(0)$ and $P(n) \implies P(n+1)$.

For proofs with programming languages, we generalize the idea of mathematical induction to **structural induction**. Until this point, we've done proofs about individual programs, e.g. above showing that a particular concrete expression has a particular type. As you've seen, these have an inductive flavor—to prove a statement, you recursively prove statements about its sub-components until you reach a base case.

While the proofs we've done have been about *concrete* expressions, the next step is to generalize our proofs to work on *arbitrary* expressions. For example, if we want to prove a proposition P on all well-typed expressions, then we have to prove the proposition holds for *all the ways a type can be constructed for an expression*. This is fairly abstract, so let's dive into the progress/preservation proofs to get an example of what this looks like.

Proving type safety

Recall the preservation theorem: if $e : \tau$ and $e \mapsto e'$ then $e' : \tau$. To prove this for our simply typed lambda calculus, we are going to proceed by structural induction over the different ways a type can be constructed, i.e. each typing rule in our static semantics (also called "rule induction"). The typing rule will tell us a more specific version of the proposition to prove, and also provide us with certain facts from our inductive hypothesis.

> Note: the progress and preservation theorems are defined with respect to "closed terms", i.e. expressions which don't need a type context at the top level to prove their type. Or put another way, expressions with no free variables.

Proof. By rule induction on the static semantics.

1. T-Var: if $x : \tau$ and $x \mapsto e'$ then $e' : \mathsf{int}$.

   This is vacuously true, since a variable x cannot have a type τ without a typing context Γ.

2. T-Int: if $n : \mathsf{int}$ and $n \mapsto e'$ then $e' : \mathsf{int}$.

   This is vacuously true, since there is no rule to step a number n.

3. T-Binop: if $e_1 \oplus e_2 : \mathsf{int}$ and $e_1 \oplus e_2 \mapsto e'$ then $e' : \mathsf{int}$.

First, by the premises of the T-Binop rule, we know $e_1 : \mathsf{int}$ and $e_2 : \mathsf{int}$.

Second, by the inductive hypothesis (IH), we get to assume preservation holds true for e_1 and e_2. For example, if $e_1 \mapsto e_1'$ then we know $e_1' : \mathsf{int}$ (and likewise for e_2).

Third, we case on the three ways in which a binary operation can step:

A. D-Binop-1: assume $e_1 \mapsto e_1'$, so $e_1 \oplus e_2 \mapsto e_1' \oplus e_2$. By the IH, $e_1' : \mathsf{int}$, so by T-Binop we have that $e_1' \oplus e_2 : \mathsf{int}$.

B. D-Binop-2: assume e_1 val and $e_2 \mapsto e_2'$, so $e_1 \oplus e_2 \mapsto e_1 \oplus e_2'$. By the IH, $e_2' : \mathsf{int}$, so by T-Binop we have that $e_1 \oplus e_2' : \mathsf{int}$.

C. D-Binop-3: assume $e_1 = n_1$ and $e_2 = n_2$ and $n_3 = n_1 \bar{\oplus} n_2$, so $n_1 \oplus n_2 \mapsto n_3$. By T-Int we have that $n_3 : \mathsf{int}$.

Hence, in every case, we have shown that $e' : \mathsf{int}$ for all possible e', and the preservation theorem holds for T-Binop.

4. T-Lam: if $(\lambda\ (x : \tau_1)\ .\ e) : \tau_1 \to \tau_2$ and $\lambda\ (x : \tau_1)\ .\ e \mapsto e'$ then $e' : \tau_1 \to \tau_2$.

   This is vacuously true, since there is no rule to step a function value.

5. T-App: if $e_1\ e_2 : \tau$ and $e_1\ e_2 \mapsto e'$ then $e' : \tau$.

   First, by the premises of T-App, we know $e_1 : \tau_1 \to \tau_2$ and $e_2 : \tau_1$.

   Second, by the IH, we know that preservation holds for e_1 and e_2.

   Third, we case on the two ways an application can step:

   A. D-App-1: assume $e_1 \mapsto e_1'$, so $e_1\ e_2 \mapsto e_1'\ e_2$. By the IH, $e_1' : \tau_1 \to \tau_2$, so by T-App, we know $e_1'\ e_2 : \tau_2$.

   B. D-App-2: assume $e_1 = \lambda\ (x : \tau)\ .\ e_1'$, so $e_1\ e_2 \mapsto [x \to e_2]\ e_1'$.

      By the inversion[6] of T-Lam, we know $\tau = \tau_1$ and $x : \tau_1 \vdash e_1' : \tau_2$.

      By the substitution typing lemma[7], $x : \tau_1 \vdash e_1' : \tau_2 \implies [x \to e_2]\ e_1' : \tau_2$.

   Hence, preservation holds in either case.

Since preservation holds for all typing rules, then it holds for the entire language. ■

Lastly, let's prove progress.

Theorem. if $e : \tau$ then either e val or there exists e' such that $e \mapsto e'$.

Proof. By rule induction on the static semantics.

1. T-Var: if $x : \tau$ then either x val or there exists e' such that $e \mapsto e'$.

   This is vacuously true, since a variable x cannot have a type τ without a typing context Γ.

2. T-Int: if $n : \mathsf{int}$ then either n val or there exists e' such that $n \mapsto e'$.

   By D-Int, n val.

3. T-Binop: if $e_1 \oplus e_2 : \mathsf{int}$ then either $e_1 \oplus e_2$ val or there exists e' such that $e_1 \oplus e_2 \mapsto e'$.

   First, by the premises of the T-Binop rule, we know $e_1 : \mathsf{int}$ and $e_2 : \mathsf{int}$.

   Second, by the inductive hypothesis (IH), we get to assume progress holds true for e_1 and e_2. For example, if either e_1 val or $e_1 \mapsto e_1'$.

   Third, we case on the different possible states of e_1 and e_2 derived from the IH:

   A. $e_1 \mapsto e_1'$: then by D-Binop-1, $e_1 \oplus e_2 \mapsto e_1' \oplus e_2$.

   B. e_1 val $\wedge\ e_2 \mapsto e_2'$: then by D-Binop-2, $e_1 \oplus e_2 \mapsto e_1 \oplus e_2'$.

   C. e_1 val $\wedge\ e_2$ val: because $e_1 : \mathsf{int}$ and $e_2 : \mathsf{int}$, then by inversion on D-Int we know $e_1 = n_1$ and $e_2 = n_2$. Therefore by D-Binop-3, $n_1 \oplus n_2 \mapsto n_3$ for $n_3 = n_1 \bar{\oplus} n_2$.

In each case, the expression steps, so progress holds.

4. T-Lam: if $(\lambda\ (x : \tau_1)\ .\ e) : \tau_1 \to \tau_2$ then either $(\lambda\ (x : \tau_1)\ .\ e)$ val or there exists e' such that $(\lambda\ (x : \tau_1)\ .\ e) \mapsto e'$.

   By D-Lam, $(\lambda\ (x : \tau_1)\ .\ e)$ val.

5. T-App: if $e_1\ e_2 : \tau$ then either $e_1\ e_2$ val or there exists e' such that $e_1\ e_2 \mapsto e'$.

   First, by the premises of T-App, we know $e_1 : \tau_1 \to \tau_2$ and $e_2 : \tau_1$.

   Second, by the IH, we know that progress holds for e_1 and e_2.

   Third, we case on the different possible states of e_1 derived from the IH:

   A. $e_1 \mapsto e_1'$: then by D-App-1, $e_1\ e_2 \mapsto e_1'\ e_2$

   B. e_1 val: then by inversion on D-Lam, $e_1 = \lambda\ (x : \tau)\ .\ e_1'$. By D-App-2, $e_1\ e_2 \mapsto [x \to e_2]\ e_1'$.

   In each case, the expression steps, so progress holds.

Since progress holds for all typing rules, then it holds for the entire language. ■

---

1. In the lambda calculus, all functions technically take one argument, so when I say "a function that takes one argument", I mean as opposed to a function that returns another function. ↩
2. Except where built into the language, of course. In the `div` example, both of the asserted invariants (int types and nonzero) will be checked by the division operator in the language runtime. ↩
3. The options provided do not strictly form a dichotomy. "Gradual" or "hybrid" invariant enforcement that mixes static/dynamic checks is an active area of research, e.g. gradual typing. ↩
4. Why is the arithmetic necessary? Can't we just keep our functions-only approach? Unfortunately, no. Imagine the function $\lambda\ x\ .\ x$ and I asked you: what is the type of this function? At some point, you have to have a "base type", since a type language of just Type $\tau ::= \tau_1 \to \tau_2$ is infinitely recursive, and you cannot construct an actual type. ↩
5. You can think about Γ as a "purely functional" dictionary. Adding a new mapping like $x : \tau$ will overwrite a previous mapping for x. ↩
6. Inversion is a useful proof technique/lemma to cite on occasion. Generally speaking, it falls from the rule: if $A \implies B$, and $\nexists C.\ C \implies B$, then $B \implies A$, i.e. $A \iff B$. Above, since there's only one way to construct a type for a lambda, if we have a lambda and know it has a type, then we can deduce that its body must be well typed. ↩
7. You can assume if $x : \tau \vdash e' : \tau'$ and $e : \tau$ then $[x \to e]\ e' : \tau'$. ↩