An introduction to Type Theory and Formal Semantics

Andrew Clifton

November 9, 2017

Type theory is a deductive system for reasoning about the properties of *programs* written in *programming languages*. Although the most familiar meaning of "type" is "type of value" (e.g., integer, string, etc.) type theory actually encompasses all *static properties* of programs: the type may include any information that can be reliably extracted from programs without running them. Obviously, the kinds of information available depend heavily on the language in which programs are written, hence the natural connection between type theory and programming language semantics. Type theory also offers an alternate foundation from which mathematical structures can be specified, by viewing types as propositions, and values as proofs of those propositions.

Background

Type theory originated in the design of *strongly-typed* programming languages. In a strongly-typed language, every value, and hence every expression, has a fixed *type*: for example, Int(eger) or Str(ing, a textual sequence of characters). The constraints on an operation can be described in terms of the types of its input(s), and the type of its output. For example, `+` takes two Int inputs and gives an Int output. Formally, we would write this type as something like

$$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$$

Some operations "mix up" types, as in the `length` operation which gives the length of a string, which has type Str → Int. We can also imagine a `print` operation which takes an Int and gives the string consisting of its textual representation. (E.g., `print(1)` would evaluate to the string `"1"`.) This would have type Int → Str.

A *program* consists of some arrangement of operations, which must be syntactically correct (matched parentheses and so forth), but not all syntactically-correct arrangements make sense with respect to the *types*. In order for a program to make sense — to be "well-typed" — input types must align with output types. For example, the program

```
print(1 + 2)
```

is well-typed, because 1 and 2 are both Ints which is what `+` expects as inputs, and the result of `+` is an Int, which is expected by `print`. Conversely, this program

```
length(1 + "a")
```

is *ill*-typed, because `+` must have two Int inputs, but its right input is a Str. (Note that `length` has no say in the matter; once a type error has occurred, we don't bother looking anywhere else.)

We want to describe the types of the operations in this (fictional) language in such a way that we can answer the question: What is the type, if any, of a given program? (If we cannot determine the type of a program, we say that it is *ill-typed*.)

Inference rules

We're going to write typing rules as inference rules, so here's a quick introduction. An inference rule is written as a set of *premises* above a *conclusion*:

$$\frac{P \quad Q \quad R}{C}$$

This rule has three premises, P, Q, and R, and a conclusion C. Expressed informally, the rule says "*if* P, Q, and R are true, *then* C is true." Another way to think of it is as saying, "to prove the truth of C, you must prove P, Q, and R". (On the other hand, if we already know that C is true, then P, Q, and R should be true, also.)

Every inference rule must have a conclusion, but a rule may have zero premises:

$$\frac{}{C}$$

This rule is an *axiom* stating that C is always true.

Often we'll label our rules with names, so we can refer to them later:

$$\textsc{Ex}_1 \frac{P \quad Q \quad R}{C}$$

As an example, suppose we have the following rules:

$$\frac{\text{It is raining}}{\text{I'll have an umbrella}} \qquad \frac{}{\text{It is raining}}$$

What conclusions can we draw from these rules? (That it is raining, and that I'm carrying an umbrella.) Given a system of rules, we can do two things:

- We can start with the axioms and work our way *down*, seeing how many conclusions we can prove from them. Often this set will be infinite, so this technique is of limited use.
- Given some judgment whose truth is *unknown* (i.e., a proposition), we can work *upwards*, from conclusions to premises, trying to tie off every line of reasoning with an axiom. A complete "tree" of rule applications, from proposition to axioms, is called a *derivation*. Note that while the existence of a derivation implies the truth of the starting judgment, the *lack* of a derivation is not proof of its negation. For example, if it is *not*

raining, we cannot conclude that I *will not* have an umbrella; there might be other, unstated, reasons why I would have an umbrella.

Example 1 *Here is a system of rules about the primitive judgments P, Q, and R:*

$$\textsc{RuleA}\frac{}{P} \qquad \textsc{RuleB}\frac{P}{Q} \qquad \textsc{RuleC}\frac{P \quad Q}{R}$$

Using these, we can construct a proof of R:

$$\textsc{RuleC}\frac{\textsc{RuleA}\dfrac{}{P} \qquad \textsc{RuleB}\dfrac{\textsc{RuleA}\dfrac{}{P}}{Q}}{R}$$

If we have two rules with the same conclusion, then, when trying to prove it, we have two choices as to how to proceed:

$$\frac{P}{R} \qquad \frac{Q}{R}$$

To prove R we can either try to prove P, or we can try to prove Q.

Example 2 *Natural numbers can be defined by the two rules*

- *Zero is a natural number*
- *If x is a natural number, then $1 + x$ is also a natural number.*

  As inference rules, this defines a judgment nat:

$$\textsc{NatZero}\frac{}{0\ \mathsf{nat}} \qquad \textsc{NatSucc}\frac{x\ \mathsf{nat}}{1 + x\ \mathsf{nat}}$$

Given this definition, we can prove that $3\ \ (=\ 1 + 1 + 1 + 0)\ \ \mathsf{nat}$, by constructing a derivation:

$$3\ (=1{+}1{+}1{+}0)\ \mathsf{nat} \quad\longrightarrow\quad \textsc{NatSucc}\frac{\textsc{NatSucc}\dfrac{\textsc{NatSucc}\dfrac{\textsc{NatZero}\dfrac{}{0\ \mathsf{nat}}}{1 + 0\ \mathsf{nat}}}{1 + (1 + 0)\ \mathsf{nat}}}{1 + (1 + (1 + 0))\ \mathsf{nat}}$$

Note that we are *not* constructing the *set* of natural numbers, $\mathbb{N}$. Doing so would require the third property of an inductive set, *closure*, the statement that "nothing else is a natural number". What I've built here is an (intuitionistic) *judgment* defining "natural number-ness". If we try to prove something like

$$1 + \mathsf{potato}\ \mathsf{nat} \quad\longrightarrow\quad \textsc{NatSucc}\frac{\dfrac{?}{\mathsf{potato}\ \mathsf{nat}}}{1 + \mathsf{potato}\ \mathsf{nat}}$$

the answer we get is not a definitive *false*, but rather "unable to be proven".

Put another way, the judgment nat describes when something *is* a natural number; the question of when something *is not* a natural number is a completely different question, and would require a completely different judgment.

Hypothetical judgments

Later on we will need the ability to use *hypothetical judgments* in our inference rules. A hypothetical judgment is one in which we *temporarily* assume something to be true, but only in some particular branch of a derivation. We write a hypothetical judgment as

This is an intuitionistic (one consequent) presentation of hypothetical judgments, using sequents, otherwise known as Gentzen's System LJ. The other option is to use *natural deduction*, which is a bit more cumbersome for our purposes.

$$\ldots \vdash C$$

To the left of the $\vdash$ is the list of assumptions, and C is the *consequent* of the judgment. (In fact, when we write a non-hypothetical judgment C we are implicitly stating $\vdash C$; that is, C is true with *no* assumptions.)

We usually assume that hypothetical judgments are governed by the following rules (structural):

$$\textsc{Assumption}\ \frac{}{\ldots, C, \ldots \vdash C} \qquad \textsc{Exchange}\ \frac{\ldots, B, A, \ldots \vdash C}{\ldots, A, B, \ldots \vdash C} \qquad \textsc{Weakening}\ \frac{\ldots, A, A, \ldots \vdash C}{\ldots, A, \ldots \vdash C}$$

These basically work together to make the set of assumptions act like a proper set: order of assumptions doesn't matter (rule Exchange) and duplicates are ignored (rule Weakening). The Assumption rule is what lets us actually *use* an assumption: if we assume that C is true, then C is axiomatically true in that context.

Judgments

Judgments are the things that A, B, etc. stand for above. A judgment is something that can be true, or potentially proven to be true. The most important judgments in our type theory will be

Judgment	Meaning
e prog	e is a syntactically-valid program
t typ	t is a syntactically-valid type
$x : \tau$	Expression x is of type τ
$x \mapsto x'$	Expression x takes a single evaluation step to x'
v val	v is a fully-evaluated value

The first three relate to the *statics* (type semantics) of a language, while the last two relate to its *dynamics* (runtime behavior).

A specification for Int-Str

Here we will use inference rules to present a *typing specification* of the language Int-Str, a simple language with two types (Int and Str) and a few operations. We assume that the language has the following syntactical elements:

- Integer constants 0, 1, 2, etc. We will embed these into the language as $n[0], n[1], \ldots$.
- String constants, embedded as s["…"]
- The infix binary + operator on integers
- The one-argument (unary) function `length`, taking a string input and returning an integer.
- The one-argument `print` function, taking an integer input and returning a string.

In the style of nat above, we can give a similarly inductive definition of the collection of *syntactically* valid programs in Int-Str:

$$\textsc{ProgInt}\ \frac{}{n[x]\ \mathsf{prog}} \qquad \textsc{ProgStr}\ \frac{}{s[x]\ \mathsf{prog}}$$

$$\textsc{Prog+}\ \frac{e_1\ \mathsf{prog} \quad e_2\ \mathsf{prog}}{e_1 + e_2\ \mathsf{prog}} \qquad \textsc{ProgLength}\ \frac{e\ \mathsf{prog}}{\mathtt{length}(e)\ \mathsf{prog}} \qquad \textsc{ProgPrint}\ \frac{e\ \mathsf{prog}}{\mathtt{print}(e)\ \mathsf{prog}}$$

Note that prog says nothing about whether a program is well-typed; it only describes those programs which are syntactically valid. According to the above rules,

$$n[1] + s[\texttt{"a"}]\ \mathsf{prog}$$

is a syntactically-valid program; it is, however, not a well-typed one.

Statics: Types of Valid Programs

The language Int-Str only has two types, Int and Str, for simplicity. In a more complex system we might want to defined a judgment typ to define the structure of syntactically valid types, but that's not necessary yet (we will define typ when we add compound types).

Recall that to say that x is of type τ we write the judgment $x : \tau$.

The types of the constants are given by the rules

$$\textsc{TypeInt}\ \frac{}{n[x] : \mathsf{Int}} \qquad \textsc{TypeStr}\ \frac{}{s[x] : \mathsf{Str}}$$

These rules are axiomatic, specifying that *any* integer literal, and any string literal, is a well-typed program.

The remainder of the typing rules are not axiomatic: they depend on the types of their subexpressions being correct.

$$\textsc{Type+}\frac{x : \mathsf{Int} \qquad y : \mathsf{Int}}{x + y : \mathsf{Int}} \qquad \textsc{TypeLength}\frac{s : \mathsf{Str}}{\texttt{length}(s) : \mathsf{Int}} \qquad \textsc{TypePrint}\frac{x : \mathsf{Int}}{\texttt{print}(x) : \mathsf{Str}}$$

Using these rules, we can determine the type of a (valid) program, while simultaneously showing that the program is well-typed:

$$\textsc{TypePrint}\frac{\textsc{Type+}\dfrac{\textsc{TypeLength}\dfrac{\textsc{TypeStr}\dfrac{}{s[\texttt{"a"}] : \mathsf{Str}}}{\texttt{length}(s[\texttt{"a"}]) : \mathsf{Int}} \qquad \textsc{TypeInt}\dfrac{}{n[1] : \mathsf{Int}}}{\texttt{length}(s[\texttt{"a"}]) + n[1] : \mathsf{Int}}}{\texttt{print}(\texttt{length}(s[\texttt{"a"}]) + n[1]) : \mathsf{Str}}$$

On the other hand, we'll get stuck if we try this with an ill-typed program:

$$\texttt{print}(\texttt{print}(n[1]) + s[\texttt{"a"}]) : ?$$

Dynamics: Evaluation of Programs

While the "statics" of a type system describe the types of a program, the closely-related *dynamic* specification describes the evaluation (runtime behavior) of a program. This is defined using the $\mapsto$ judgment to describe how a program "takes a step", and the val judgment to describe when a program cannot take another step (has completed). Each step is a single transformation of the program, progressing towards the final val.

For Int-Str, the literal values are val; they are fully evaluated and cannot take a step:

$$\textsc{ValInt}\frac{}{n[x]\ \mathsf{val}} \qquad \textsc{ValStr}\frac{}{s[x]\ \mathsf{val}}$$

A program consisting of only an integer constant, or only a string literal, is finished.

The operations +, `length` and `print` must be defined somehow; presumably there are machine code instructions (or system subroutines) which, when executed, will perform addition, compute the length of a string, or convert an integer to its string representation. We'll represent these internal operations as the primitive operations `ADD`, `LEN`, and `PRN`. For example, the rules will specify that $n[1] + n[2] \mapsto n[\texttt{ADD}(1, 2)]$, indicating that we construct a new numeric constant from the actual sum of 1 and 2.

Let us consider the behavior of a program such as $(n[1] + n[2]) + (n[2] + n[3])$. Looking at the outer +, we cannot execute the `ADD` operation yet, because the arguments are not yet fully evaluated. Hence, we need several evaluation rules, to describe the order of operations. I will assume that the operands are evaluated left-to-right. This leads to the following evaluation

rules:

$$\textsc{Search+}_1 \frac{e_1 \mapsto e'_1}{e_1 + e_2 \mapsto e'_1 + e_2} \qquad \textsc{Search+}_2 \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1 + e_2 \mapsto e_1 + e'_2} \qquad \textsc{Plus} \frac{}{n[x_1] + n[x_2] \mapsto n[\texttt{ADD}(x_1, x_2)]}$$

If the left operand (e_1 above) can take a step, then it does so, replacing the original operand with the stepped-to value. If the left operand is val, then we perform the same procedure on the right operand. When both operands are integer constants ($n[x]$ for some x) then we actually perform the addition. Note that the PLUS rule takes effect not just when both arguments are val, but specifically when they are both *numeric*. If we had specified the rule as

$$\textsc{Plus} \frac{e_1 \text{ val} \quad e_2 \text{ val}}{e_1 + e_2 \mapsto ?}$$

then we wouldn't have been able to extract the x_1, x_2 needed to perform the ADD.

SEARCH+$_1$ and SEARCH+$_2$ are sometimes called *search* rules, because they direct the "search" for the part of the program that can actually be executed next.

We follow a similar procedure, but simplified, for length and print: evaluate the input until it is val, and then apply the primitive operation to it.

$$\textsc{SearchLen} \frac{e \mapsto e'}{\texttt{length}(e) \mapsto \texttt{length}(e')} \qquad \textsc{Len} \frac{}{\texttt{length}(s[x]) \mapsto n[\texttt{LEN}(x)]}$$

$$\textsc{SearchPrint} \frac{e \mapsto e'}{\texttt{print}(e) \mapsto \texttt{print}(e')} \qquad \textsc{Print} \frac{}{\texttt{print}(n[x]) \mapsto s[\texttt{PRN}(x)]}$$

Notice that none of these rules make any reference to the *types* Int and Str. This is known as *type erasure*: presumably we only run programs that are well-typed, so the actual typing information is not needed at runtime.

Using these rules, we can "run" a program such as

$$\texttt{length}(\texttt{print}(n[1] + n[2])) + n[3]$$

The result has a two-dimensional structure: applying the search rules constructs a derivation, whose purpose is to find the *innermost* portion of the program that can be executed. This results in a new program, on which we perform a new search, and so forth until the result is val.

Type Safety

Given a static and dynamic specification for a programming language, we would like to show that the resulting language is *safe*; i.e., that no well-typed program "goes wrong", but always finishes with a value (val) of the

program's type. This is called *type safety* and consists of two propositions:

- PROGRESS: For all $e : \tau$, either $e \mapsto e'$ *or* e val is true. (Every well-typed program either takes a step or is finished.)
- PRESERVATION: For all $e : \tau$, if $e \mapsto e'$ then $e' : \tau$. (If a well-typed program takes a step, then its type is the same before and after.)

Note that these are properties of *whole programming languages*, not individual programs. Once we have proven that a language is type-safe, then any particular program in that language obviously will be.

Proof of PROGRESS: Progress is proved by induction on typing judgment $e : \tau$. The proof will have one case for each form of the typing judgment (i.e., for each typing rule above). The proof depends on a fairly simple pair of lemmas, known as the *canonical forms* lemmas:

LEMMA 1 *If* $e : \mathsf{Int}$ *and* e val *then there exists* x *such that* $e = n[x]$.

LEMMA 2 *If* $e : \mathsf{Str}$ *and* e val *then there exists* x *such that* $e = s[x]$.

These essentially state that the vals of each type are *closed*. Thus, if we have e val and we know e's type, we can safely dismantle it to get at the x (either integer or string) inside.

PROOF For all $e : \tau$ either e val or there exists e' with $e \mapsto e'$.
By induction on $e : \tau$.

Inductive hypothesis: For any subexpression x of e, $x : \tau$ implies either x val or there exists $x', x \mapsto x'$.

- Case 1 and 2: $e = n[x] : \mathsf{Int}$ and $e = s[x] : \mathsf{Str}$. The constants are easy; constants are val and hence cannot take a step, and thus e val.
- Case 3: $e = e_1 + e_2 : \mathsf{Int}$. Our assumptions are $e_1 : \mathsf{Int}$, $e_2 : \mathsf{Int}$ (from the rule TYPE+). By the IH, we can assert that both e_1 and e_2 are either val or take steps. There are subcases for each of the possibilities:
  1. If $e_1 \mapsto {e'}_1$ then $e_1 + e_2 \mapsto {e'}_1 + e_2$ by rule SEARCH+$_1$. (Note that we don't care about e_2 in this case.)
  2. If e_1 val and $e_2 \mapsto {e'}_2$ then $e_1 + e_2 \mapsto e_1 + {e'}_2$ by rule SEARCH+$_2$
  3. If e_1 val and e_2 val then by the canonical forms lemmas there are x_1, x_2 such that $e_1 = n[x_1], e_2 = n[x_2]$, and thus $n[x_1] + n[x_2] \mapsto n[\mathtt{ADD}(x_1, x_2)]$ by rule PLUS
- Cases 4 and 5, for `length` and `print` are very similar: we gain an assumption about the type of the argument, and by the IH, the argument (a subexpression) must be either val (and hence the canonical forms lemmas apply) or take a step to some other expression. In either case, the evaluation rules give a step for the whole expression e. ⊣

Proof of PRESERVATION: Preservation is proved by induction on the step relation $e \mapsto e'$. There are cases for each step rule.

Inductive hypothesis: for any subexpression x of e, if $x : \tau'$ and $x \to x'$ then $x' : \tau'$.

- There are no cases for constants, because they cannot take a step.
- If $e = e_1 + e_2$ then by inversion of $e_1 + e_2 : \mathsf{Int}$, we know $e_1, e_2 : \mathsf{Int}$. There are three subcases one for each + rule:

  - If $e_1 \mapsto e'_1$ then by the IH, $e'_1 : \mathsf{Int}$. So then we have $e' = e'_1 + e_2$ where both are of type Int, and by rule TYPE+, $e' : \mathsf{nat}$.
  - If e_1 val, then $e_2 \mapsto e'_2$, and by the IH we have $e'_2 : \mathsf{Int}$, and $e' = e_1 + e'_2$. So again, by rule TYPE+ we have $e' : \mathsf{Int}$.
  - If $e_1 = n[x_1], e_2 = n[x_2]$ then $e' = n[\mathtt{ADD}(x_1, x_2)]$ and by rule TYPEINT, $e' : \mathsf{Int}$.

- If $e = \mathtt{length}(e_1)$ then by inversion of $\mathtt{length}(e_1) : \mathsf{Int}$, we know $e_1 : \mathsf{Str}$. Once again, there are two cases, for the two `length` evaluation rules:
  - If $e_1 \to e'_1$ then by IH, $e'_1 : \mathsf{Str}$, so $\mathtt{length}(e'_1) : \mathsf{Int}$ by rule `TypeLength`.
  - If $e_1 = s[x]$ for some x, then $e' = n[\mathtt{LEN}(x)]$ and by rule `TypeInt`, $e' : \mathsf{Int}$.
- The case for `print` is similar; either the inner expression takes a step (and then by IH it's type is preserved), or the primitive operation is used.

  Some other interesting properties of Int-Str are

- DETERMINISM OF $\mapsto$: Prove that every non-val program $e : \tau$ takes a step to a *unique* program e'. The easiest way to state this property is

  $$\text{if } e \mapsto e_1 \text{ and } e \mapsto e_2 \text{ then } e_1 = e_2$$

- TERMINATION: Prove that every well-typed program e, if stepped enough times, will eventually terminate (become val). If we define the iteration of $\mapsto$, $\mapsto^*$

  $$\frac{e \text{ val}}{e \mapsto^* e} \qquad \frac{e \mapsto e' \quad e' \mapsto^* e''}{e \mapsto^* e''}$$

  this amounts to proving that for every $e : \tau$, there is some e' such that $e \mapsto^* e'$ and e' val.

Proof of these is left as an exercise for the reader.

Variable bindings

Variables, which can be *bound* to values, are a common feature of programming languages and will serve as a stepping-stone to fully-fledged functions. To add variable bindings to Int-Str we will add a new operation: `let`. The syntax of `let` is

$$\mathtt{let}\ v = e\ \mathtt{in}\ e'$$

and has the effect of binding the variable v to the value e, within the subexpression e'. For example:

$$\mathtt{let}\ v = n[1]\ \mathtt{in}\ v + v$$

would evaluate to 2. Outside of the `in` part, the variable v does not exist, and its use is not valid. (Note that the entire `let`…`in`…is a *single* operation.)

The syntactical and typing rules for `let` are somewhat subtle, as they require the use of hypothetical judgments. We will consider the syntax rule prog first: when is a `let`-expression syntactically valid? It may be helpful to examine some *non*-syntactically valid `let`-programs:

- $\texttt{let}\ v = n[1]\ +\ \texttt{in}\ v + v$
  Here, the program is invalid because the expression assigned to v is invalid.

- $\texttt{let}\ v = n[1]\ \texttt{in}\ v\ +$
  Here, the program is invalid because the main expression is invalid.

- $v + n[1]$
  Here the program is invalid because a variable v is used *outside* the main expression of a corresponding `let`.

Obviously the expression assigned to the variable (e in $v\ =\ e$) must be valid on its own. Thus, one of our conditions is $e\ \ \mathsf{prog}$. And outside of a `let`, v is *not* valid, but it *is* valid within e'. We express this by requiring that $e'\ \ \mathsf{prog}$ *assuming* that $v\ \ \mathsf{prog}$. v is not a valid program universally, but only as part of "check" on the validity of e'.

We can express these conditions using hypothetical judgments:

$$\textsc{ProgLet}\ \frac{e\ \mathsf{prog} \qquad v\ \mathsf{prog} \vdash e'\ \mathsf{prog}}{\texttt{let}\ v = e\ \texttt{in}\ e'\ \mathsf{prog}} \qquad \text{where } v \text{ is } \textit{fresh}$$

The extra condition on the end serves to avoid difficulties that arise when we have `let`s inside other `let`s. We require that each `let` use a "fresh" variable, so there is no possibility of collision with variable names. It is always possible to rename variables so that no name is reused. Note that v is only assumed to exist within e', and not within e; this rules out "infinite" bindings such as

$$\texttt{let}\ v = v\ \texttt{in}\ n[1] + v$$

Using this definition, we can prove that a simple `let`-expression is syntactically-valid:

$$\textsc{ProgLet}\ \frac{\textsc{ProgInt}\ \dfrac{}{n[1]\ \mathsf{prog}} \qquad \textsc{Assumption}\ \dfrac{}{v\ \mathsf{prog} \vdash v\ \mathsf{prog}}}{\texttt{let}\ v = n[1]\ \texttt{in}\ v\ \mathsf{prog}}$$

However, if we try to prove this of some more complex `let`, we'll get

stuck:

$$\textsc{ProgLet}\dfrac{\textsc{ProgInt}\ \dfrac{}{n[1]\ \mathsf{prog}} \qquad \dfrac{?}{v\ \mathsf{prog} \vdash v + v\ \mathsf{prog}}}{\mathtt{let}\ v = n[1]\ \mathtt{in}\ v + v\ \mathsf{prog}}$$

The problem is that none of the *other* prog rules allow for assumptions. E.g., the PROG+ rule

$$\textsc{Prog+}\dfrac{e_1\ \mathsf{prog} \qquad e_2\ \mathsf{prog}}{e_1 + e_2\ \mathsf{prog}}$$

can only be applied when there are no assumptions ($e_1 + e_2$ prog implicitly means $\vdash e_1 + e_2$ prog). In order to allow the rest of the system to work with `let`, we need to thread the current context of assumptions through all the rules. We will use Γ to represent all the assumptions in the context. This leads to the following re-definition of prog:

$$\textsc{ProgInt}\dfrac{}{\Gamma \vdash n[x]\ \mathsf{prog}} \qquad \textsc{ProgStr}\dfrac{}{\Gamma \vdash s[x]\ \mathsf{prog}}$$

$$\textsc{Prog+}\dfrac{\Gamma \vdash e_1\ \mathsf{prog} \qquad \Gamma \vdash e_2\ \mathsf{prog}}{\Gamma \vdash e_1 + e_2\ \mathsf{prog}}$$

$$\textsc{ProgLength}\dfrac{\Gamma \vdash e\ \mathsf{prog}}{\Gamma \vdash \mathtt{length}(e)\ \mathsf{prog}} \qquad \textsc{ProgPrint}\dfrac{\Gamma \vdash e\ \mathsf{prog}}{\Gamma \vdash \mathtt{print}(e)\ \mathsf{prog}}$$

$$\textsc{ProgLet}\dfrac{\Gamma \vdash e\ \mathsf{prog} \qquad \Gamma, v\ \mathsf{prog} \vdash e'\ \mathsf{prog}}{\Gamma \vdash \mathtt{let}\ v = e\ \mathtt{in}\ e'\ \mathsf{prog}}$$

(Notice that PROGLET is the only rule which *modifies* the context, adding an assumption about the validity of its variable.)

Now if we try to prove the validity of

$$\textsc{ProgLet}\dfrac{\textsc{ProgInt}\ \dfrac{}{\vdash n[1]\ \mathsf{prog}} \qquad \textsc{Prog+}\ \dfrac{\textsc{Assumption}\ \dfrac{}{v\ \mathsf{prog} \vdash v\ \mathsf{prog}} \qquad \textsc{Assumption}\ \dfrac{}{v\ \mathsf{prog} \vdash v\ \mathsf{prog}}}{v\ \mathsf{prog} \vdash v + v\ \mathsf{prog}}}{\vdash \mathtt{let}\ v = n[1]\ \mathtt{in}\ v + v\ \mathsf{prog}}$$

we'll be successful; the context will be carried *into* the subexpressions of + where it can be used.

Statics of Let-Int-Str

Adding `let` to the static specification requires a similar effort: what is the type of `let` v `=` e `in` e'? Clearly, it depends on the type of e: the type of e' is what results when e's type is "injected" into it, at every point where v is

used. Once again, we will use hypothetical judgments to *assume* that v's type is the same as e's, while we are typing e':

$$\textsc{TypeLet}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma, v : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{let}\ v = e\ \texttt{in}\ e' : \tau'}$$

Just as with prog, we need to integrate Γ into the other typing rules:

$$\textsc{TypeInt}\ \frac{}{\Gamma \vdash n[x] : \mathsf{Int}} \qquad \textsc{TypeStr}\ \frac{}{\Gamma \vdash s[x] : \mathsf{Str}}$$

$$\textsc{Type+}\ \frac{\Gamma \vdash x : \mathsf{Int} \qquad \Gamma \vdash y : \mathsf{Int}}{\Gamma \vdash x + y : \mathsf{Int}} \qquad \textsc{TypeLength}\ \frac{\Gamma \vdash s : \mathsf{Str}}{\Gamma \vdash \texttt{length}(s) : \mathsf{Int}} \qquad \textsc{TypePrint}\ \frac{\Gamma \vdash x : \mathsf{Int}}{\Gamma \vdash \texttt{print}(x) : \mathsf{Str}}$$

Note that under these rules a `let` expression is only well-typed if its bound expression (e) is well typed. This is true, even if the bound expression is never used! For example,

$$\texttt{let}\ v = n[1] + s[\texttt{"a"}]\ \texttt{in}\ n[2]$$

is not well-typed, even though we know exactly the type it will always have: Int. This restriction is necessary because we need to know the type of $e : \tau$ for the assumption $v : \tau$.

Dynamics of `let`

When a `let` expression is evaluated, what happens? In order to define the effect result of $\texttt{let}\ v = e\ \texttt{in}\ e' \mapsto \cdots$, we will need to define *substitution*.

Definition 3 A *substitution* of the expression x for the variable v in e is written

$$[x/v]e$$

and has the effect of replacing every occurrence of v in e with x.

It's possible to completely define the effect of a substitution via inference rules; e.g.,

$$\frac{}{[x/v]v = x}$$

$$\frac{[x/v]e_1 = e'_1 \qquad [x/v]e_2 = e'_2}{[x/v](e_1 + e_2) = e'_1 + e'_2}$$

and so forth. This amounts to "threading" the substitution through all the syntactical constructs.

We have two options for the evaluation rules for `let`:

- We can substitute the expression bound to v *immediately*, whether or not it is val. This leads to the rule

$$\textsc{Let}\ \frac{}{\texttt{let}\ v = e\ \texttt{in}\ e' \mapsto [e/v]e'}$$

This is known as *call-by-name* binding semantics.

- We can first step the bound expression all the way to val, and only then

do the substitution. This leads to the pair of rules

$$\textsc{SearchLet}\frac{e_1 \mapsto e_2}{\texttt{let}\ v = e_1\ \texttt{in}\ e \mapsto \texttt{let}\ v = e_2\ \texttt{in}\ e} \qquad \textsc{Let}\frac{e\ \mathsf{val}}{\texttt{let}\ v = e\ \texttt{in}\ e' \mapsto [e/v]e'}$$

This is known as *call-by-value* semantics.

If the variable v is evaluated multiple times, then call-by-value is more efficient; if v is evaluated only once, or not at all, then call-by-name can be more efficient. (Under the rules given, both the by-name and by-val evaluation schemes will always produce the same results, just with potentially-differing numbers of steps needed to get there. Proving this is an interesting exercise.)

You might be wondering what happens, under call-by-value, when we have a `let` whose bound expression e includes a variable from an outer `let`:

$$\texttt{let}\ x = n[1]\ \texttt{in}\ (\texttt{let}\ y = x + n[1]\ \texttt{in}\ y + y)$$

How can we step $x + n[1]$ until val? The answer is that we don't have to! The outer `let` will step until $n[1]$ val (which is immediate), and then perform the substitution. By the time we start stepping the inner `let`, x will have been replaced by $n[1]$ and we can step $n[1] + n[1]$ to val.

Proving the type-safety properties (progress and preservation) in the presence of `let` involves a bit of extra work. For preservation, we must prove a lemma stating that substitution correctly preserves types; that is

$$\text{if } v : \tau \vdash e' : \tau' \text{ then } [e/v]e' : \tau'$$

Usually we just assume that this lemma holds, as the sort of things that tend to "break" it (such as destructive variable assignment) don't appear in any of our rules.

Functions

A `let` can be thought of as a single-argument function where we supply the actual argument at the same time as the function definition. Adding functions to our language allows us to separate function *definition* from *application*.

When we start to implement functions, we immediately have a decision to make: do we want *named* functions, à la $f(x) = x + x$, or *un-named* functions? Interestingly, the latter are *easier* to specify than the former.

Function abstractions

An *unnamed function* or λ-abstraction is a *value* which represents a computation. The function f above can be written in λ form as

$$\lambda(x : \mathsf{Int}).x + x$$

The general pattern is $\lambda(v : \tau).b$ where v is the variable that stands for the function's eventual argument (which must be of type τ), and b is an expression, called the *body* of the function, which may refer to v, and which the function will expand into when applied. The syntax of a λ includes the argument type, but not the return type, as that can be inferred.

Specifying the type of a function requires us to add a *function type*. We write the type of functions that take τ_1 as an input, and return a τ_2 as

$$\tau_1 \to \tau_2$$

For example,

$$\lambda(x : \mathsf{Str}).\mathtt{length}(x) : \mathsf{Str} \to \mathsf{Int}$$

$\to$ is a *compound* type, one built out of smaller types. Note also that the type of the function input is part of the syntax of an abstraction.

To add functions, we need to add rules for prog, typing, and evaluation. The prog rule for functions is similar to that for `let`, except that the expression e is missing:

$$\textsc{Prog}\lambda\ \frac{\tau\ \mathsf{typ} \qquad \Gamma, v\ \mathsf{prog} \vdash b\ \mathsf{prog}}{\Gamma \vdash \lambda(v : \tau).b\ \mathsf{prog}}$$

A λ is syntactically-valid if its body b is, under the assumption that v is (and if the type given for v is a syntactically-valid typ).

The typing rule for λ is as follows:

$$\textsc{Type}\lambda\ \frac{\Gamma, x : \tau_1 \vdash b : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).b : \tau_1 \to \tau_2}$$

This states that a λ is a function from τ_1 to τ_2 if assuming that its variable is of type τ_1 causes its body to be of type τ_2.

The evaluation rule for λ is simple: functions are always val.

$$\textsc{Val}\lambda\ \frac{}{\lambda(x : \tau).b\ \mathsf{val}}$$

Because we now have compound types, we should in theory construct a judgment typ to specify when a *type* is syntactically-valid:

$$\textsc{TypInt}\ \frac{}{\mathsf{Int\ typ}} \qquad \textsc{TypStr}\ \frac{}{\mathsf{Str\ typ}}$$

$$\textsc{Type}\to\ \frac{\tau_1\ \mathsf{typ} \qquad \tau_2\ \mathsf{typ}}{\tau_1 \to \tau_2\ \mathsf{typ}}$$

Function application

Functions are "created" by *abstraction*, by building a λ. Functions are used by *application*, by supplying a valid to be substituted for the variable. We write application of f to the expression e as just $f\ e$.

Here are the syntax, statics, and dynamics of function application:

$$\textsc{ProgApp}\ \frac{\Gamma \vdash f\ \mathsf{prog} \qquad \Gamma \vdash e\ \mathsf{prog}}{\Gamma \vdash f\ e\ \mathsf{prog}}$$

$$\textsc{TypeApp}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

For the dynamics there are once again both call-by-name:

$$\textsc{SearchApp}\frac{f \mapsto f'}{f\ e \mapsto f'\ e} \qquad \textsc{App}\frac{}{(\lambda(x:\tau).b)\ e' \mapsto [e'/x]b}$$

where the argument e is substituted into b without being evaluated, and call-by-value, where e is also evaluated all the way to val before application:

$$\textsc{SearchApp}_1\frac{f \mapsto f'}{f\ e \mapsto f'\ e} \qquad \textsc{SearchApp}_2\frac{f\ \mathsf{val} \qquad e \mapsto e'}{f\ e \mapsto f\ e'} \qquad \textsc{App}\frac{e\ \mathsf{val}}{(\lambda(x:\tau).b)\ e \mapsto [e/x]b}$$

In both cases, we must step the function f until we get a λ-abstraction: only by looking at the full $\lambda(x:\tau).b$ do we know what the variable and body of the function are.

If we have λ-abstractions, we don't actually need `let` as a separately-defined construct; we can define `let` purely in terms of λ:

$$\texttt{let}\ v = e\ \texttt{in}\ e' \ \equiv\ (\lambda(v:\tau).e')\ e \qquad\qquad (\text{if } e:\tau)$$

Looking forward: other types

Function types are just the one kind of compound types we can add to our system.

Product and sum types

A product type is like a `struct` in C/C++: it takes elements of two (or more) types and wraps them up into a single value. The product type is usually written $\tau_1 \times \tau_2$; values of this type are written as $\langle e_1, e_2 \rangle$. The type rule for $\times$ is just

$$\textsc{Type}\times\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

In addition to the product *constructor*, we also add two *extractor* functions, Ex_1 and Ex_2, used to extract the two components from an existing product:

$$\textsc{TypeEx}_1\frac{\Gamma \vdash p : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{Ex}_1(p) : \tau_1} \qquad \textsc{TypeEx}_2\frac{\Gamma \vdash p : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{Ex}_2(p) : \tau_2}$$

The evaluation rules for products state that a product is a val if both its components are, and that Ex_1 applied to a product takes a step to its *left* component, while Ex_2 takes a step to its *right* component. (Similarly to the by-value/by-name choice for function application, here we can choose whether $\mathrm{Ex}_{1,2}$ take a step immediately, or only after their component is val, or only after *both* components are val.)

The extractors have the following types

$$\text{Ex}_1 : \tau_1 \times \tau_2 \to \tau_1 \qquad \text{Ex}_2 : \tau_1 \times \tau_2 \to \tau_2$$

If we reverse the directions of the arrows we get

$$\text{InL} : \tau_1 \to \tau_1 + \tau_2 \qquad \text{InR} : \tau_2 \to \tau_1 + \tau_2$$

+ is known as a *co-product* or *sum* type. While the product of τ_1 and τ_2 includes *both* types, the sum of τ_1 and τ_2 include *one or the other* of the two, but never both. The functions InL and InR are called *injectors*.

Technically, we need to attach the full sum type to both injectors, so that we know what the other "half" of the sum is. So the injectors should actually be something like $\text{InR}_{\tau_1,\tau_2}$.

Having built up a value of a sum type, how do we use it? Because we don't know which "half" of the sum it might be (τ_1 or τ_2) we must supply code to handle *both*. The result is `case`, a kind of if-else structure that branches based on the type of value stored in the sum:

$$\textsc{TypeCase}\ \frac{\Gamma \vdash s : \tau_1 + \tau_2 \qquad \Gamma, v_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, v_1 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}(s, v_1.e_1, v_2.e_2) : \tau}$$

`case` takes an expression of sum type and two expressions in which the variables v_1, v_2 are bound, either to a value of type τ_1 or of τ_2. In either case, e_1 and e_2 must have a final type of τ.

As an example of how this might be used, suppose we wanted to write a function `mag` which can operate on the sum of Int and Str, returning the "magnitude" of its input. For integers this means the value, while for strings this means length. The `mag` function looks like this:

$$\mathtt{mag} \equiv \lambda(x : \mathsf{Int} + \mathsf{Str}).\mathtt{case}(x, v_1.v_1, v_2.\mathtt{length}(v_2))$$

If x was created by InL then we'll have $v_1 = x$ and the `case` will evaluate to v_1. If x was created by InR then $v_2 = x$ and the `case` will evaluate to $\mathtt{length}(v_2)$. In either case, the result type of the branch is Int.

Polymorphic types

In our current system, there are infinitely many *distinct* identity functions (functions that return their input unchanged). For Int and Str the identity functions are

$$\lambda(x : \mathsf{Int}).x \qquad \text{and} \qquad \lambda(x : \mathsf{Str}).x$$

Notice that the *bodies* of body functions are identical. There are (infinitely) many more identity functions for the various function types. E.g., here is the identity function for Int → Str:

$$\lambda(x : \mathsf{Int} \to \mathsf{Str}).x$$

Once again, the body is unchanged; only the type of x changes.

We would like to extend our type system so that there is only *one* identity function, which is *polymorphic* (does the same thing for different types). To do so, we need to introduce type-level variables, variables which can contain types. This requires adding three new constructs:

- A new kind of type, $\forall\alpha.\tau$ which specifies that α is a type variable within the type τ. This is the type-level equivalent to a λ-abstraction.
- A new kind of *abstraction*, Λ, abstracting out the type from a body of code.
- A new kind of *application*, $e[\tau]$ which applies the Λ abstraction e to the type τ.

(Interestingly, just as we needed to add contexts to prog when we added `let` and function, we will need to add contexts to typ now that we have type-level functions.)

The system with λ functions and polymorphic types is known as the *polymorphically-typed lambda calculus* or *Girard's System **F***.

A surprising "feature" of $\forall$ types is that once we have them, we don't need sum or product types to be built-in; we can build them ourselves! In fact, it's even possible to construct the Int type out of $\forall$ types. System F is remarkably expressive in its type system, able to define a large variety of types that would otherwise need to be specified "by hand".

Another surprising property of polymorphic types is how a $\forall$ type actually gives us a significant amount of information about its elements. For example, the type $\forall\alpha.\alpha \to \alpha$ has only *one* element, the polymorphic identity function:

$$\Lambda\ t.\lambda(x : t).x$$

No other value of the type $\forall\alpha.\alpha \to \alpha$ can exist. To understand why, imagine that $\forall\alpha.\alpha \to \alpha$ describes a machine which, if you provide it with an object of some unknown type α, will give you back another α. Crucially, the machine is *not* allowed to do anything that would require knowledge of the "real" type of α. Given this restriction, the only thing the machine can do is give you back the original α you provided it with; it can't build a *new* α, because it has no idea how α's are built. If something has type $\forall\alpha.\alpha \to \alpha$ we immediately know that it must be the poly. identity function, without even looking at its definition; that's all it can be.