CPSC 311: Typed Fun **(DRAFT)**
(“lec-typed-fun”)

Joshua Dunfield
University of British Columbia

October 12, 2016

1 Defining a type system

We didn’t specify any static semantics for Fun, so Fun is untyped. It’s time to design *Typed* Fun.

This will enable us to catch many errors in Fun programs statically (after parsing) rather than during evaluation. It will also let us write type annotations (signatures) in Fun programs that are actually checked.

Again, a language is syntax *and* semantics. Typed Fun will have almost the same syntax as Fun; the only change will be (in the abstract syntax) variants for type annotations. The important difference will be in the static semantics.

When we make our language typed, how will it affect the dynamic semantics? We’ll find out!

2 Code

The code for these notes can be found in

`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/notes/typing-lam.rkt`

3 When does typing happen?

Typing is part of *static* semantics, and evaluation is part of *dynamic* semantics, so the type checker has to run before the interpreter. In most language implementations, type checking is done after parsing. We can view the parser and type checker as increasingly restrictive filters: the parser only accepts programs that match a BNF, and the type checker only accepts programs that are well-typed (that can be typed according to a system of rules).

Some “advanced” type systems are implemented not as replacements of the type checker, but as additional type checkers that are run after the “normal” one. For example, the SML-CIDRE “sort checker” is run after the usual SML type checker, providing a *third* filter on what counts as a valid program.

4 Are ill-typed programs meaningful?

Should we regard a program that isn’t well-typed as having any meaning at all?

Taking the definition of a language to be “syntax + static semantics + dynamic semantics”, then a reasonable answer is “no”: Just as we don’t try to give any kind of semantics (static or dynamic) to strings that are not accepted by the parser, we shouldn’t try to give a dynamic meaning to programs that are not accepted by the type checker.

However, we could interpret the question as, “If I imagine a *different* language that’s the same but with no type system, does the program have a meaning?” In that case, there is no type system to filter out programs, and we can certainly ask what the *dynamic* meaning of the program is; we've been doing this for [*Un*typed] Fun for weeks now!

A tricky but more interesting version of the question arises in languages that can have *more than one type* for each expression, that is, when $\Gamma \vdash e : A$ and $\Gamma \vdash e : B$ and $A \neq B$. For example, in Java, every object is a subclass of `Object`, and so a variable of type `Integer` has both the type `Integer` and the type `Object`. The original question could be rephrased as, "Does an ill-typed program have zero or one meanings?" Now the question becomes, "Does a program with more than one type have more than one meaning?" The answer is a matter of taste. We could say e has *two* meanings, A and B, or that the (static) meaning of e is not one type, but the set of types $\{A, B\}$. Whether we choose to say that e has two static meanings or one, we still need to relate the static meaning(s) of e to the dynamic meaning of e (what it evaluates to).

For now, all of our type systems will have the property that each expression has at most one type. But you should keep in mind that this is a property of these particular type systems, not of all type systems.

In the case of Java, we could have only one static meaning by saying that the meaning of an expression is its smallest type (that is, its "lowest" class). This is sometimes called the *principal type* of the expression.

5 Defining a type system

We didn't specify any static semantics for Fun, so Fun is untyped. It's time to design *Typed* Fun.

This will enable us to catch many errors statically (after parsing) rather than during evaluation. It will also let us write type annotations (signatures) in Fun programs that, unlike signatures in comments in Racket, are actually checked.

Again, a language is syntax *and* semantics. Typed Fun will have almost the same syntax as Fun; the only change will be (in the abstract syntax) variants for type annotations. The important difference will be in the static semantics.

When we make our language typed, does it affect the dynamic semantics? We'll find out!

5.1 Typing judgment

The usual judgment form for whether an expression is typed, and which type it has, is

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

The symbol $\vdash$ is called a *turnstile*; it separates the judgment into assumptions on the left, and something that—if the whole judgment is derivable—is a logical consequence of the assumptions. (Thus, at a very high level, it means "implies"; but that does not distinguish it from many other notions in logic and programming languages, including Gentzen's horizontal lines.)

Ignoring the Γ for the moment, the judgment says that the given expression e has the type A. In the dynamic semantics of Fun, the operators (+, =, etc.) evaluate only if they are applied to appropriate kinds of animals (numbers vs. booleans vs. functions), so our type system should, at minimum, distinguish numbers and booleans. We would then expect

$$\Gamma \vdash (\mathsf{Num}\ 3) : \mathsf{num}$$

and

$$\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}$$

to be derivable.

Just these two types num and bool would suffice for a language without functions. But to handle functions, or even just Let, the judgment form needs to talk about the types of the expression's free identifiers. Our dynamic semantics ($e \Downarrow v$) didn't need assumptions, because those rules substituted away identifiers. So, assuming we start with a program e that is closed (has no free identifiers), we never need to evaluate (or step) any expression with free identifiers.

On the other hand, types are *static* semantics—meanings given to expressions without evaluating them. Given a function (Lam x e), we can't substitute a value for x, because we don't know what value to use until the function is applied, and we don't know how the function will be applied without evaluating the whole program.

To give a *static* meaning—a type—to a function body, or to the body of a Let, we need to handle expressions that have free identifiers. Such expressions are called *open*, because they are not closed.

The assumptions Γ, also called a *typing context*, simply list the types of the free identifiers. As we recently did for values and evaluation contexts (for small-step semantics), we can use a BNF grammar to define what a Γ is. As with values and evaluation contexts, we are *not* specifying concrete syntax. In fact, we're going one step further away from using a BNF for concrete syntax, because Γ has no direct connection to the syntax of the language. Rather, we are using the BNF notation for a different purpose.

$$\begin{array}{llll} \text{Typing contexts} & \Gamma ::= & \emptyset & \text{empty context (no assumptions)} \\ & & \mid x : A, \Gamma & x \text{ has type } A\text{, with more assumptions} \end{array}$$

This BNF defines a list of assumptions: you can think of $\emptyset$ as the empty list, and $x : A, \Gamma$ as the "cons" of a head $x : A$ and a tail Γ. Given this BNF, we should not be able to write

$$x : A$$

alone for a typing context with a single assumption; rather, we should write

$$x : A, \emptyset$$

By convention, however, typing contexts are treated somewhat more informally, so that $x : A$ and $x : A, y : B$ would be considered valid contexts. However, we will follow the BNF exactly when we encode it as a **define-type**. *This* BNF represents abstract syntax, not as concrete syntax, so we encode it using **define-type**. We don't need to write a parser for Γ because Γ won't appear in programs, only in the typing judgment.

5.2 Typing for AE (arithmetic expressions)

To design the typing rules, we can "replay" most of the development that led us to Fun: start with Num, Add and Sub; add Let; add functions; add recursion; add booleans and Ite; add pairs.

Until we reach Let, we won't actually need Γ—it will always be empty—but I'm adding it now to smooth the transition to the larger language.

The rules for AE expressions are given in Figure 1. There are only three rules... and only one type! Having only one type isn't very useful. (These rules are, in a sense, only repeating the structure of the AE abstract syntax itself; if you read ": num" as "is an AE", the rules become a less compact notation for **define-type**.) When we put booleans back, we'll add a second type bool, and then the types will really tell us something. In fact, the types will matter earlier than that...

$\boxed{\Gamma \vdash e : \mathsf{num}}$ Under assumptions Γ (always empty, here), AE expression e has type num

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{AE-Type-num}$$

$$\frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Add}\ e1\ e2) : \mathsf{num}}\ \text{AE-Type-add} \qquad \frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Sub}\ e1\ e2) : \mathsf{num}}\ \text{AE-Type-sub}$$

Figure 1 Typing rules for AE expressions

5.3 Typing

We can add Let and Id to the above language. Now, Γ will serve a purpose. We type the body of a Let using a new assumption $x : \mathsf{num}$, and when we type $(\mathsf{Id}\ x)$, we check that "$\Gamma(x) = \mathsf{num}$", that is, that $x : \mathsf{num}$ appears somewhere in Γ.

It would be possible, but tedious, to say that $\Gamma(x) = \mathsf{num}$ is another form of judgment, and write rules deriving it.

$\boxed{\Gamma \vdash e : \mathsf{num}}$ Under assumptions Γ (not always empty, now!), AEL expression e has type num

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{AELType-num}$$

$$\frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Add}\ e1\ e2) : \mathsf{num}}\ \text{AELType-add} \qquad \frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Sub}\ e1\ e2) : \mathsf{num}}\ \text{AELType-sub}$$

$$\frac{\Gamma \vdash e : \mathsf{num} \qquad x : \mathsf{num}, \Gamma \vdash eBody : \mathsf{num}}{\Gamma \vdash (\mathsf{Let}\ x\ e\ eBody) : \mathsf{num}}\ \text{AELType-let} \qquad \frac{\Gamma(x) = \mathsf{num}}{\Gamma \vdash (\mathsf{Id}\ x) : \mathsf{num}}\ \text{AELType-var}$$

Figure 2 Typing rules for AEL expressions

We still only have one type, but our typing rules are no longer useless: they are checking that all free variables in the expression appear in Γ. When we begin typing an expression, we are not inside any Let expression, so Γ is $\emptyset$; as we enter the body of a Let, we add the variable now in scope.

Consequently, if we *can* derive $\emptyset \vdash e : \mathsf{num}$, then evaluating e *cannot* raise a free-variable-error. For example, evaluating $(\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ x))$ will raise a free-variable-error because y is free, but that expression is rejected by typing: we cannot derive

$$\emptyset \vdash (\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ x)) : \mathsf{num}$$

■ **Exercise 1.** Try to derive the above judgment.

■ **Exercise 2.** Derive $y : \mathsf{num} \vdash (\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ x)) : \mathsf{num}$.

■ **Exercise 3.** Derive $\emptyset \vdash \big(\mathsf{Let}\ y\ (\mathsf{Num}\ 2)\ (\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ x))\big) : \mathsf{num}$.

(If you derived a judgment in a previous exercise, and that judgment appears as a premise, just write a checkmark above the premise.)

Since we only have one type, we have no "agreement errors", but the type system does catch the one kind of error we have so far.

5.4 AEL + booleans

To add booleans (Bfalse, Btrue, Ite), we need more than one type, so it's time to give a BNF grammar for types as well. Whether this BNF grammar is serving as concrete syntax (like the grammar for $\langle \mathtt{E} \rangle$) or another notation for **define-type** can be set aside for now, because the grammar is so simple; we'll have to revisit this later, much to my annoyance.

$$\begin{array}{llll} \text{Types} & A, B ::= & \mathsf{num} & \text{numbers} \\ & \mid & \mathsf{bool} & \text{booleans} \end{array}$$

Compared to AEL, some of the rules don't change at all, and some are completely new (for Bfalse, Btrue, Ite). Type-let and Type-var have the same structure, but instead of the single type num everywhere, these rules have meta-variables A and B, allowing Let to bind a num in a body of type bool, or a bool in a body of type num, or any other combination.

Writing A and B in Type-let doesn't mean that A and B are necessarily different types, only that they don't have to be the same type. If we wanted to require A and B to be different, we would need to add a premise $A \neq B$.

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, AEL+booleans expression e has type A

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{Type-num}$$

$$\frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Add}\ e1\ e2) : \mathsf{num}}\ \text{Type-add} \qquad \frac{\Gamma \vdash e1 : \mathsf{num} \qquad \Gamma \vdash e2 : \mathsf{num}}{\Gamma \vdash (\mathsf{Sub}\ e1\ e2) : \mathsf{num}}\ \text{Type-sub}$$

$$\frac{}{\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\ \text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{Btrue}) : \mathsf{bool}}\ \text{Type-true}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Let}\ x\ e\ eBody) : B}\ \text{Type-let} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) : A}\ \text{Type-var}$$

Figure 3 **Typing rules for expressions for AEL+booleans**

■ **Exercise 4.** In assignment 2, we replaced Add and Sub with Binop and then added Lessthanop and Equalsop. Suppose we decide not to use Binop, but instead add an abstract syntax variant Lessthan that behaves like a2's (Binop (Lessthanop) ...). Design a rule "Type-Lessthan" that types expressions of the form (Lessthan $e1$ $e2$).

5.5 All the typing rules (that we can't implement)

Adding rules for functions (Type-lam and Type-app) below is pretty straightforward on paper. Unfortunately, we can't implement Type-lam! The problem is that our type checker is only given Γ and e. We know that e has the form $(\mathsf{Lam}\ x\ e\mathsf{Body})$, so we know that we need to apply rule Type-lam, but we don't know what A is, so we don't know what we need to derive next!

This leads us to the concept of the *mode* of a meta-variable.

(Aside: In the rule Type-binop, I am assuming a judgment form $op : A1{*}A2 \rightarrow B$, read "operator op takes two arguments of types $A1$ and $A2$, respectively, and returns a value of type B".)

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \text{Type-var}$$

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\ \text{Type-num} \qquad \frac{op : A1 * A2 \rightarrow B \qquad \Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Binop}\ op\ e1\ e2) : B}\ \text{Type-binop}$$

$$\frac{}{\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\ \text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{Btrue}) : \mathsf{bool}}\ \text{Type-true}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e\mathsf{Then} : A \qquad \Gamma \vdash e\mathsf{Else} : A}{\Gamma \vdash (\mathsf{Ite}\ e\ e\mathsf{Then}\ e\mathsf{Else}) : A}\ \text{Type-ite}$$

$$\frac{x : A, \Gamma \vdash e\mathsf{Body} : B}{\Gamma \vdash (\mathsf{Lam}\ x\ e\mathsf{Body}) : A \rightarrow B}\ \text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \rightarrow B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}\ \text{Type-app}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash e\mathsf{Body} : B}{\Gamma \vdash (\mathsf{Let}\ x\ e\ e\mathsf{Body}) : B}\ \text{Type-let}$$

Figure 4 Typing rules for Typed Fun... not implementable!

5.5.1 One judgment, many problems

Each judgment form is defined by the rules that can derive it, that is, the rules that have that judgment form as the rule's conclusion. Both philosophically and practically, each judgment form exists independently from its *implementation* as an evaluator, or stepper, or type checker. A more subtle point is that each judgment has several different implementations that solve totally different problems!

For example, our interpreter implements the evaluation judgment $e \Downarrow v$. But the problem our interpreter solves—or, equivalently, the question that running our interpreter answers—is really

"Given an expression e, is there some value v such that $e \Downarrow v$?" (Interpreter question)

We could ask a different question about that judgment:

“Given a value v, is there some expression e such that $e \Downarrow v$?” (Reverse interpreter question)

This is an easier question—it can be answered using a single line of Racket code. However, a similar-looking question is much harder to answer than the interpreter question!

“Given a value v, list *all* the expressions e such that $e \Downarrow v$.” (Horrible question)

(I’m not sure that question is even decidable... there are certainly too many such expressions to have any hope of listing them all!)

We could also ask

“Given an expression e and a value v, is the judgment $e \Downarrow v$ derivable?” (Validation question)

Here, the instantiations of both meta-variables are given.

At the other extreme, we could ask

“Do there exist e and v such that $e \Downarrow v$ is derivable?” (Vacuousness question)

Answering this question is easy: pick a rule with no premises, like Eval-num, and choose any n you like. (But it’s not a completely pointless question. Mathematicians tell a story about a PhD student who proved many interesting theorems about certain manifolds, those with properties $P1$, $P2$, $P3$, ... At the end of the student’s defence talk, a certain professor pointed out that the class of manifolds had no inhabitants, because some of the properties were mutually contradictory. According to the story, the professor phrased this question in an especially obnoxious way: “Isn’t your entire dissertation vacuous for the following trivial reasons?” If a judgment cannot be derived, it serves no purpose. However, a much more common problem in defining programming languages is that too many, or too few, judgments can be derived—not that *no* judgments of a given form can be derived.)

Forgetting about the horrible question, which is different from the other four because it asks for *all* possible instantiations of a meta-variable rather than just one, we find that a single judgment with two meta-variables e and v gives rise to $2 \times 2 = 4$ different problems, according to whether

- e is given or not given
- v is given or not given

Rephrasing the above questions (again, forgetting the horrible one):

1. The program that answers the question when e is given and v is not is an interpreter.
2. The program that answers the question when v is given, and e is not, doesn’t seem very useful, but we could call it a “reverse interpreter”.
3. The program that answers the question when both e and v are given is a program that *validates* whether a particular evaluation is correct.
4. The program that answers the question when neither e nor v is given is (hopefully) trivial, but it tells you that the judgment form isn’t vacuous (assuming at least one $e \Downarrow v$ judgment is in fact derivable).

5.5.2 Modes

If the instantiation of a meta-variable is given, we say its *mode* is *input*, and if it is not given, we say its mode is *output*. We can mark each meta-variable in a judgment form with the mode of that meta-variable. The "interpreter question" corresponds to the *moded judgment form*

$$e_{\mathsf{IN}} \Downarrow v_{\mathsf{OUT}}$$

and the "validation question" corresponds to

$$e_{\mathsf{IN}} \Downarrow v_{\mathsf{IN}}$$

(In some logic programming languages, modes can be declared with $+$ and $-$; the IN mode corresponds to $+$, and the OUT mode corresponds to $-$).

For other judgment forms, we can also list various moded judgment forms. The small-step judgment form $e1 \longrightarrow e2$ is usually moded as

$$e1_{\mathsf{IN}} \longrightarrow e2_{\mathsf{OUT}}$$

but it's reasonable to think about

$$e1_{\mathsf{OUT}} \longrightarrow e2_{\mathsf{IN}}$$

which corresponds to "expansions" in Church's λ-calculus, whereas the $e1_{\mathsf{IN}} \longrightarrow e2_{\mathsf{OUT}}$ form corresponds to Church's "reductions".

For the typing judgment $\Gamma \vdash e : A$, the problem we've been considering (and playing with in Racket during lecture) corresponds to

$$\Gamma_{\mathsf{IN}} \vdash e_{\mathsf{IN}} : A_{\mathsf{OUT}}$$

But other "modings" have been studied as well:

- The moding $\Gamma_{\mathsf{IN}} \vdash e_{\mathsf{IN}} : A_{\mathsf{IN}}$ arguably is more appropriately called type *checking* than $\Gamma_{\mathsf{IN}} \vdash e_{\mathsf{IN}} : A_{\mathsf{OUT}}$, which could be called type *inference*.
- An implementation of the moding $\Gamma_{\mathsf{OUT}} \vdash e_{\mathsf{IN}} : A_{\mathsf{OUT}}$ would try to find a type *and* a context, which is both theoretically interesting and potentially useful: imagine a type error message that says, "I can't type this, but *if only* this unknown identifier had a particular type...". For example, you could ask such an implementation to come up with a context Γ and type A such that

  $$\Gamma \vdash (\mathsf{Ite}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ z)) : A$$

  and it might suggest $\Gamma = x : \mathsf{bool}, y : \mathsf{num}, z : \mathsf{num}$ and $A = \mathsf{num}$:

  $$x : \mathsf{bool}, y : \mathsf{num}, z : \mathsf{num} \vdash (\mathsf{Ite}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ z)) : \mathsf{num}$$

  which would be nice, especially when learning a language.

  I know of one attempt to implement this, which I regard as a failure because it was extremely complicated, but perhaps there are simpler approaches.
- Finally, the moding $\Gamma_{\mathsf{IN}} \vdash e_{\mathsf{OUT}} : A_{\mathsf{IN}}$ asks: under Γ, does *anything* have type A? (In some type systems, you can write down a type that no expression has.[1])

[1]However, in languages that have a type called `void`, `void` is usually *not* such a type, causing endless grumbling among academic researchers.

6 Declarative vs. algorithmic

The difficulty we've encountered with Type-lam (which we would encounter with Type-rec, as well) represents a gap between a rule that "looks good on paper" and one that can be directly implemented. Rules that *can* be directly implemented are called *algorithmic*: if you have enough practice, you can "read off" an algorithm from the rules. We have done this rather implicitly, but all of our implementations so far—of both evaluation and typing—have depended on being able to identify, without too much effort, which rule should be used. Usually only one rule has an expression of the right form in the conclusion: if the expression is (Bfalse), only Eval-bfalse (for evaluation) or Type-bfalse (for typing) applies. Sometimes, two rules might apply, as with Eval-ite-true and Eval-ite-false; there, our interpreter uses the result of evaluating the scrutinee to decide which rule to use.

Until now (with Type-lam), we have not had to implement a rule in which a premise has missing information. Such rules are common in language definitions, or at least in the more theoretical work that underlies some languages (especially typed languages); type systems are connected to logics, but logicians are usually more interested in the theoretical properties of their rules than the connections to type systems. (This is historically true, at least, and logicians who predated the development of computers can hardly be faulted for not focusing on those connections!)

Rules that cannot be directly implemented, whether because of missing information or some other difficulty, are called *declarative*: they "declare" what the judgment means, but not "algorithmically". A common tactic of programming languages researchers is to define a simple and (hopefully) understandable "declarative type system", and *then* define an "algorithmic version" that *looks* very different but accepts exactly the same programs as the declarative type system.

We will not pursue that tactic now. Instead, we will change the definition of expressions in a way that provides the missing information in Type-lam (and in Type-rec). This isn't my favourite solution, but it's one that (I think) fits the time we have for Assignment 3; it's also a solution that's closer to what languages like C and Java do than the solution I like better.

7 Typing rules we can implement

The problem we have is that we don't know what A is in Type-lam. So we'll put A *into the expression*. The concrete syntax for Lam will have a ⟨Type⟩, and the **define-type** branch will have an extra argument containing the domain of the function (the type of its argument); we can figure out the range of the function (the type of its result) by looking at the function body.

$$\frac{x : A, \Gamma \vdash e\mathsf{Body} : B}{\Gamma \vdash (\mathsf{Lam}\ x\ \boxed{A}\ e\mathsf{Body}) : A \to B}\ \text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \to B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}\ \text{Type-app}$$

We also have to do this for Type-rec, which makes me sad, because my preferred (but harder to explain) solution wouldn't make us do this.

$$\frac{u : B, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Rec}\ u\ \boxed{B}\ e) : B}\ \text{Type-rec}$$

Since a Rec usually has a Lam as its body, this means we have to write the function domain twice: once in the Rec, as part of a function type, and once in the Lam.

```
{Rec u {-> num bool}
     {Lam x num
          {Ite {= x 0}
               Btrue
               {Ite {= x 1}
                    Bfalse
                    {Ite {< x 0}
                         {App u {+ x 2}}
                         ; x must be >1
                         {App u {- x 2}}}}}}}
```