Definition of Programming Languages: A Principled Approach in Racket

Jana Dunfield
University of British Columbia (2014–2017)
Queen's University (2017–?)

December 22, 2021

Contents

Preface

This document contains lecture notes from the 2016W1 instance of CPSC 311 at UBC, with a few small changes and additions.

1 Introduction

1.1 World domination

"Definition of Programming Languages"? What is this course about?

It might be about world domination. For the first 50 years or so of programming languages (if we take Fortran, in the mid-1950s, as the beginning; though we should acknowledge the plans for the Analytical Engine of Charles Babbage and Ada Lovelace in the 19th century, as well as Konrad Zuse's *Plankalkül* in the 1940s), particular languages were expected to achieve some form of world domination. Every one of these languages failed to completely dominate, but this didn't seem to dampen the hopes of advocates of the next world-dominating language:

- 196x: Algol was going to dominate
- 1970: PL/I was going to dominate
- 1980: C was going to dominate
- 1990: C++ was *totally* going to dominate (see the paper on "Oak", in which James Gosling explained how he had to justify inventing a new language at all, instead of just using C++)
- 2000: Java was going to dominate

We might, at last, be learning that world domination is not so readily achieved. I'm not entirely sure why this is; the proliferation of the Internet and the web may be one reason: it's relatively clear that the language you write web pages in should be different from the language used to write the web server. JavaScript, for all its flaws, has at least cemented the idea that languages other than C, C++, and Java exist.

1.2 Unimpressed by fads

Since world domination seems off the table, this course will not be impressed by fads; it will not focus on one or two currently-popular languages. This course will also not be a "trip to the zoo" where you learn a little about a large number of languages. Instead, we will focus on concepts and methods that underlie (what I think of as) good programming languages. Good ideas get adopted... but it takes time. A lot of time. Automatic memory management (garbage collection), which frees programmers from deallocating memory by hand (and perhaps deallocating it twice, among many other bugs), was pioneered by Lisp in the 1960s; it was adopted in a "mainstream" language, Java, in the 1990s.

This process may be speeding up: the Rust language (backed by Mozilla) has ideas and technologies invented only 10–15 years ago.

1.3 Course goals

You will learn how to

- **understand** design choices (scope, evaluation order, types...) and some arguments for (and against) them;
- **understand**, **modify**, and **reason about** definitions of programming languages;
- **implement** interpreters for programming languages.

You will *not* learn how to write a compiler (that's CPSC 411), though much of what you learn in this course is useful for that.

This course is a magic-free zone, because programming languages aren't magic. They're still lots of fun! Studying PLs is about scaling up from programming—where you may remember realizing that *you can tell the computer what to do*—to telling the computer *how to understand the instructions*.

1.4 Definition of "Programming Languages"

What is a programming language? Agreement on a precise definition is elusive, but for this course, we will define a programming language as: a well-defined way to instruct computers, using symbols.

If computers compute (do computations), then a programming language is a **precise**, **symbolic** description of a set of possible computations.

Caveats:

- "Symbolic": there have been occasional attempts at visual PLs (Smalltalk-80 and Logo are not really visual, but languages like Prograph, developed in the 1980s and 1990s, were certainly intended to be visual).
- "Precise" is often "aspirational".

The second caveat is unfortunate:

- **Programmers** need precision so they know what programs are supposed to do.
- Language **implementors** need precision so they know how to implement (interpret, compile, translate to another language) a language.
- Unfortunately, most PLs are defined using English; a few are defined using math/logic.
- Unclear what **can** be defined, and what **should** be defined: see "The C language does not exist" from *Communications of the ACM*.

A key idea in programming language research is that there are deep connections between (some) PLs and (some) **logics**.

A programming language is a system of computation; a logic is a system of reasoning. (Just as there are many programming languages, for different purposes, there are many logics, depending on what you want to reason about.) A proof of "if X, then Y" is like a function of type $X \to Y$: given an X, it produces a Y. We'll probably only touch on this in 311.

So how do we actually define a programming language?

- **Syntax** describes *which sequences of symbols are reasonable*.
- **Dynamic semantics** describes *how to run programs*.
- **Static semantics** describes *what programs are*.

1.4.1 Syntax

Of the three parts of a language definition—syntax, dynamic semantics, and static semantics—syntax is (usually) the easiest to define, understand, and process. Using Racket makes it even easier than usual. (This was an accident: the inventors of Lisp designed a more complex syntax, but the simple syntax had already spread. For once, simplicity won.)

We won't spend much time on syntax.

1.4.2 Semantics

Dynamic semantics is about *how* programs behave:

- Dynamic semantics tells you how to "step" a program.
- You can't ride a bus effectively unless you know that buses tend to move forward.

Static semantics is about *what* programs are.

- Static semantics tells you how to understand a program **without** stepping it.
- You don't want to experimentally ride every bus until you get where you want to be. ("See where it takes you"?!)

Rules define how to step a program:

$$\frac{v1 \in \mathbb{Z} \qquad v2 \in \mathbb{Z} \qquad n = v1 + v2}{(+\ v1\ v2) \longrightarrow n} \qquad \frac{e1 \longrightarrow e2}{(v\ e1\ \ldots) \longrightarrow (v\ e2\ \ldots)}$$

Here we have two rules. The things above the line are *premises*, and the things below the line are *conclusions*. The first rule says that if $v1$ and $v2$ are integers, and n is what you get by adding $v1$ and $v2$, then the expression $(+\ v1\ v2)$ "steps to" n.

If this reminds you of the "laws of computation" from *How to Design Programs*: BSL Intermezzo, it should! It's essentially the same idea.

1.4.3 Static semantics

A [static] **type system** keeps out sort-of-nonsense:

```
(+ "no" 1)
```

Like stepping, type systems can be defined by rules.

$$\frac{e1 : \text{number} \qquad \ldots \qquad en : \text{number}}{(+\ e1\ \ldots\ en) : \text{number}}$$

This rule says that if evaluating $e1$ gives you a number, and evaluating $e2$ gives you a number, and... evaluating en gives you a number, then adding $e1 \ldots en$ together also gives you a number.

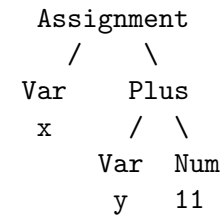
2 Syntax

2.1 Bird's-eye view

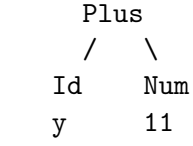
As discussed in the introduction, syntax describes "which sequences of symbols are reasonable". Given an input string, the first thing done by interpreters and compilers is to *parse* the string into an *abstract syntax tree* (AST).

Parsing is not the focus of this course, but we need to spend a little time on it. (If you take 411, parsing will probably be covered in somewhat more detail.)

For example, in a Java program, the string `x = y + 11;` would be parsed into something like



The Racket expression `(+ y 11)` could be parsed into something like

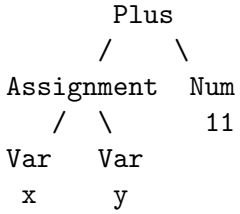


(This is not how DrRacket actually handles syntax, but it's close enough for now.)

Parsing filters out strings that don't make any sense in the language. A Java compiler, for example, will reject `(+ y 11)` with a syntax error.

The area of computer science called *formal languages* tends to focus on filtering; for example, formal languages theorists try to study which classes of automata can *recognize* strings, that is, decide whether or not a string is syntactically well-formed. In that setting, a "language" is just a set of strings. But in *programming* languages, we almost always care about the *meaning* of a particular string, so the main purpose of parsing is to get an abstract syntax tree.

It can be quite tricky to transform a string into an abstract syntax tree. Languages in the Lisp family, including Racket, make the problem easier by making the syntax unusually simple. For example, the precedence rules for Java say that + has higher precedence than =, that is, + "binds tighter". If + had *lower* precedence than =, then the Java statement `x = y + 11;` would produce the following:



In Racket, the issue just doesn't arise: the parentheses in `(+ y 11)` are *not* optional. (The rough equivalent to the Java statement, in Racket, would be `(set! x (+ y 11))`.)

2.2 Phases of parsing

Parsing is usually a two-step process:

1. *Lexical analysis* (also called *lexing* or *tokenizing*) turns a string into a sequence of *tokens*.

   For example, the Java string `x = y + 11;` would become a sequence of 6 tokens:

   id(x), equals, id(y), plus, num(11), semicolon

   This is the *only* thing lexical analysis does. It will not reject the string `x y = + + 11 + ;`—it is syntactically invalid, but it is a sequence of valid Java tokens, which is the only thing this step cares about.

   Note that lexical analysis ignores comments and whitespace (spaces, tabs, and newline characters) between tokens: the Java string

   ```
   x= /* hi */ y+11    ;
   ```

   would also become the above sequence of tokens.

   In the vast majority of languages, whitespace *inside* a token is usually not allowed: the Java string

   ```
   x= /* hi */ y+1 1    ;
   ```

   is a different sequence of tokens, with two num(1) tokens rather than one num(11) token.

   But whitespace inside some tokens, like *string literals*, is allowed: in Java (and C),

   ```
   x= "ab cd ef";
   ```

   is a sequence of 4 tokens, where the 3rd token is string(`ab cd ef`).

2. The second step is called *parsing*. (Confusingly, "parsing" can mean either the lexer and parser together, or just this second step.) This step turns a sequence of tokens into an abstract syntax tree.

Techniques for lexing are well-developed. Parsing is more difficult, but again, techniques have been developed—along with tools such as yacc, bison, and ANTLR that automatically generate parsers from *grammars*.

But we will not need to go into the details of either step. As we build interpreters, we will use DrRacket's built-in lexer/parser to do most of the work. This means that the languages we interpret must have syntax that looks a lot like Racket; this may not be to your taste, but it's less work, and leaves more time for us to discuss the more interesting aspects of programming languages.

Unfortunately, there is a gap between what DrRacket gives us, and what we need. This is the gap between *concrete syntax* and *abstract syntax*. First, we need to explain grammars.

2.3 Grammars

A grammar specifies which strings are syntactically valid programs. Different people use different notations for grammars, but most notations are based on "Backus Normal Form", or BNF, which was first used to specify the syntax of Algol-60.

Our notation looks like this:

```
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨integer⟩ ::= ⟨digit⟩
            | ⟨digit⟩⟨integer⟩
```

In this grammar, ⟨digit⟩ and ⟨integer⟩ are *nonterminal symbols*. A nonterminal on the left, like ⟨digit⟩, expands to one of the alternatives on the right. The alternatives are separated by vertical bars |. So this grammar says that a digit can have the form 0, 1, 2, . . . , and that an integer is either a single digit (⟨digit⟩), *or* ("|") a digit followed by an integer (⟨digit⟩⟨integer⟩).

Alternatives are usually written on separate lines, but for something like ⟨digit⟩ it's better to put all the alternatives on a single line.

Nonterminal symbols are usually called nonterminals, and alternatives are sometimes called *productions*.

Exercise 1. Extend the above grammar with a nonterminal ⟨nlz⟩ that represents integers *without* leading zeroes, so that ⟨nlz⟩ can have the form 13 or 130 but not 013, 00130, etc. However, your grammar *should* allow ⟨nlz⟩ to have the form 0.

Hint: First, add a nonterminal that represents a single digit that is *not* zero.

The above grammar is not terribly interesting. In fact, it is a "regular" grammar, and we could have used something simpler than BNF, like regular expressions. But we can extend this grammar to something more interesting, like "arithmetic expressions" ⟨ae⟩ in prefix notation:

```
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨integer⟩ ::= ⟨digit⟩
            | ⟨digit⟩⟨integer⟩

⟨ae⟩      ::= ⟨integer⟩
            | {+ ⟨ae⟩ ⟨ae⟩}
            | {- ⟨ae⟩ ⟨ae⟩}
```

Recalling data definitions from CPSC 110, an ⟨integer⟩ is shaped like a list: one thing, a digit, is followed by the thing being defined (an integer). In contrast, an ⟨ae⟩ is shaped like a tree, with integers as leaves, and two kinds of branches: + and -.

Remark. Mathematically speaking, BNF grammars are a particular kind of inductive definition: the nonterminal ⟨integer⟩ is defined by a base case (the alternative "⟨digit⟩") and an inductive case—the alternative "⟨digit⟩⟨integer⟩", which mentions the nonterminal ⟨integer⟩. Induction is a form of recursion: the definition of ⟨integer⟩ uses the definition of ⟨integer⟩. The nonterminal ⟨ae⟩ is also defined recursively, with two inductive (recursive) cases instead of one.

2.3.1 Grammars and abstract syntax

Following the pattern from the Java and Racket examples at the beginning of this chapter, the abstract syntax for the arithmetic expression

```
{+ 3 11}
```

could look something like

```
      Plus
      /   \
 Integer   Integer
    3         11
```

I say "could", because other variations are possible. We might call the leaf nodes `Num` rather than `Integer`, or call +'s node `Add` rather than `Plus`. We'll encounter some more interesting variations later.

The fact that such variations are possible means there has to be some "distance" between an input string like `{+ 3 11}` and its abstract syntax. For example, the grammar we wrote didn't tell us what name to give nodes for +. However, there is a kind of syntax tree that *is* uniquely determined by the grammar: a concrete syntax tree.

2.3.2 Grammars and concrete syntax

Here is the same grammar again:

$$
\begin{array}{rcl}
\langle \texttt{digit} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\langle \texttt{integer} \rangle & ::= & \langle \texttt{digit} \rangle \\
 & & \mid \langle \texttt{digit} \rangle \langle \texttt{integer} \rangle \\
\langle \texttt{ae} \rangle & ::= & \langle \texttt{integer} \rangle \\
 & & \mid \texttt{\{+} \langle \texttt{ae} \rangle \langle \texttt{ae} \rangle \texttt{\}} \\
 & & \mid \texttt{\{-} \langle \texttt{ae} \rangle \langle \texttt{ae} \rangle \texttt{\}}
\end{array}
$$

Instead of thinking about parsing a string and getting a tree, let's try to produce the string `{+ 3 11}` *from* the grammar. We can do this by replacing the left-hand sides (nonterminal symbols like ⟨ae⟩) with right-hand sides (alternatives like {+ ⟨ae⟩ ⟨ae⟩}).

1. Start with the nonterminal ⟨ae⟩.
2. We are trying to produce a string that looks like {+ ...}, so we use the second alternative (production), {+ ⟨ae⟩ ⟨ae⟩}. Now we have the string of symbols

   {+ ⟨ae⟩ ⟨ae⟩}

3. We are trying to produce {+ 3 11}, so we replace the first occurrence of ⟨ae⟩ with the first alternative of ⟨ae⟩, which is ⟨integer⟩. Now we have the string of symbols

   {+ ⟨integer⟩ ⟨ae⟩}

4. We want to produce {+ 3 ...}. The number 3 has one digit, so we replace ⟨integer⟩ with its first alternative, which is ⟨digit⟩. That gives us

   {+ ⟨digit⟩ ⟨ae⟩}

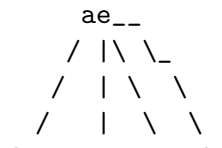
5. We specifically want 3, so we replace ⟨digit⟩ with the alternative 3:

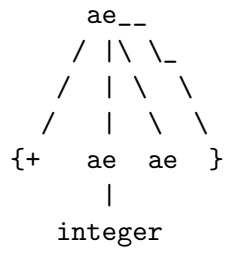
   {+ 3 ⟨ae⟩}

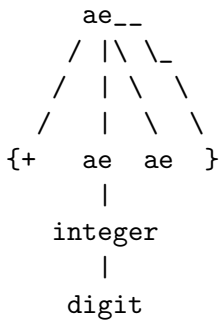
6. (Condensing some steps now.) We want the second number in the string to be `11`, so we replace ⟨`ae`⟩ with ⟨`integer`⟩, then ⟨`integer`⟩ with ⟨`digit`⟩⟨`integer`⟩, then ⟨`digit`⟩ with `1`, then ⟨`integer`⟩ with `digit`, and finally `digit` with `1`:

   `{+ 3 11}`

Here's where the "tree" part of "concrete syntax tree" comes in: instead of only writing out the string, we can build a tree as we go. Instead of replacing the nonterminal with the symbols on a right-hand side, we add those symbols to the nonterminal, as its children:







2.3.3 Tokens and nonterminals

In the grammar for ⟨`ae`⟩ above, I specified everything using the grammar—even what an integer looks like. Often, language specifications will define smaller pieces of syntax separately from bigger pieces of syntax. In that case, we would define an integer separately from an arithmetic expression, and each integer would be a single token; the main grammar would treat each token as an atom. It would look the same as the previous grammar for ⟨`ae`⟩, with the first definitions removed:

⟨`ae`⟩ ::= ⟨`integer`⟩
| `{+` ⟨`ae`⟩ ⟨`ae`⟩`}`
| `{-` ⟨`ae`⟩ ⟨`ae`⟩`}`

This style of grammar is compatible with reusing DrRacket's lexical analyzer; we'll let it handle the small pieces of syntax, like numbers. We will also use the part of DrRacket's parser that handles parentheses.

Our parser will only have to convert an *S-expression* into an abstract syntax tree. That raises new questions:

- What is an S-expression?
- How do we represent an abstract syntax tree?

2.4 S-expressions

An interesting feature of Racket (inherited from Scheme, which inherited it from Lisp) is that Racket code is also a data structure that can be directly manipulated in Racket. That data structure is called an S-expression; an S-expression is a tree that represents a Racket expression. Saying that an S-expression "represents" a Racket expression is a little questionable; it's probably more accurate to say that an S-expression *is* a Racket expression.

When we type something like

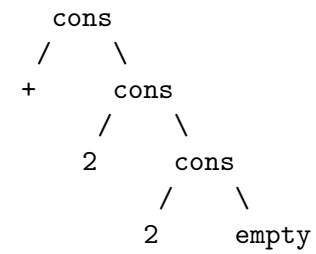
```
(+ 2 2)
```

into DrRacket, two things happen.

The first thing is that the sequence of characters

"(", "+", " ", "2", " ", "2", ")"

is parsed into an S-expression that looks like this:



S-expressions with this shape—a right-leaning tree of "cons cells", ending with `empty`—are called lists.

The second thing that happens is that the S-expression is *evaluated*. DrRacket evaluates this particular S-expression to `4`.

But you can stop DrRacket from doing the second thing: you can *quote* the expression `(+ 2 2)`, by writing an apostrophe `'` before it:

```
> '(+ 2 2)
'(+ 2 2)
```

DrRacket is still doing the first step of converting the sequence of characters into an S-expression. You can't really tell that it's doing this, because it just converts the S-expression back into a string of characters, but it is happening.

```
> '(+ 2 2)
'(+ 2 2)
```

Racket has a built-in function called `eval` that does the evaluation step, even if you've quoted an expression:

```
> (eval '(+ 2 2))
4
```

(Aside: what happens if you type the same thing *without* the quote?)

3 Operational semantics

What do programs mean? They mean whatever the[1] language definition says they do. So the real question is: How do we specify, in a language definition, the meaning of the language's programs?

Remark. I'm starting to use the lecture notes from 2015, updating as I go. In addition to outright mistakes, there may be references that are out of date, such as notes that something was done in lecture—meaning a 2015 lecture. I'll try to fix these before lecture, but I will miss some; please point them out (a note on Piazza is a good method).

3.1 Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.
- The assignment is due 1 day **before** the drop deadline.
- If you can't finish the assignment, you should probably drop the course.
- Run `handin` early and often!
  (Even if you haven't done a single problem yet!)

3.2 Dynamic semantics

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to "step" a program.
- Or how to "evaluate" a program.
- These methods work a little differently, but they have the same purpose: they tell you what your interpreter is supposed to do.

- 1. **Syntax** describes **which sequences of symbols are reasonable**.
- 2. **Dynamic semantics** describes **how to run programs**.
- 3. **Static semantics** describes **what programs are**.

[1] We'll assume that we know *which* language the program is written in, despite programs such as `http://ideology.com.au/polyglot/polyglot.txt`.

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to "step" a program.
- Or how to "evaluate" a program.
- These methods work a little differently, but they have the same purpose: they tell you what your interpreter is supposed to do.
- "Interpreter semantics":
  "to explain a language, write an interpreter for it." . . . "When we finally have what we think is the correct representation of a language's meaning. . . "
- If the interpreter you write defines the language, you **cannot** know whether it's correct. (It's trivially correct, because it defines itself. "When the President does it, that means it is not illegal.")
- You can test it on programs, but tests can only show the presence of bugs, not their absence!

3.3 Why dynamic semantics?

Unlike syntax, where practically all language designers[2] uses some variation of BNF grammars, specifying which syntactically well-formed programs actually mean something, and what they mean, is less settled.

Various methods have been used, with names like "axiomatic semantics", "operational semantics", "natural semantics", and "denotational semantics". Within the programming languages research community, there is lively competition amongst these methods. To most of the world, though, this competition is off the radar: most languages' semantics are specified informally. (Standard ML is probably the most popular formally defined language—and Standard ML is even less "mainstream" than Scheme/Racket.)

In 311, we will focus on one method, *operational semantics*. Given a specification in operational semantics, it is relatively easy (compared to specifications using other methods) to write an interpreter. Operational semantics has a rich mathematical foundation, which you would want to understand to do research in programming languages, but you don't need to understand that foundation to turn operational semantics into interpreters.

The idea of trying to specify the meaning of a mathematical object (a program) through natural language alone, rather than more "formally" (through logic and mathematics), calls to mind a quotation:

> "About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead." —Edsger Dijkstra

Formal mathematical language is not an absolute guarantee against mistakes or oversights in defining the semantics (a serious mistake in SML's formal definition went unnoticed for years), but it, at least, gives us a point of reference. Rules in natural language are for people, not computers; understanding a programming language shouldn't require one to be a "language lawyer".

[2] One exception: the designers of Algol 68, who tried to innovate in this area; it didn't end well.

3.4 Evaluation semantics

As I mentioned, operational semantics is closer to an interpreter than other methods for specifying what a program does. There are different flavours of operational semantics; we'll start with the one that's usually easier to understand, called *evaluation semantics*.

(That is, evaluation semantics is one kind of operational semantics, which is one method for specifying dynamic semantics. But for now, just remember: what we're going to do, right now, is called evaluation semantics.)

The idea of evaluation semantics is that the dynamic behaviour—the dynamic "meaning" of a program—is *the value it computes*, or equivalently, *what it evaluates to*. We expect that {+ 2 2} will compute 4, or equivalently, will evaluate to 4.

For our very first example of evaluation semantics, we'll follow the language "AE" of arithmetic expressions (from Chapter 2 of PLAI). Its concrete syntax (BNF grammar) is:

$$\begin{aligned}\langle \texttt{AE} \rangle \ ::= \ & \langle \texttt{num} \rangle \\ & \mid \{+ \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\} \\ & \mid \{- \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\}\end{aligned}$$

And its abstract syntax (PLAI, p. 6) is:

```
(define-type AE
 [Num (n number?)]
 [Add (lhs AE?) (rhs AE?)]
 [Sub (lhs AE?) (rhs AE?)])
```

However, we'll use the concrete syntax to write the evaluation semantics. This allows programmers to use our semantics to understand the language, provided they can read evaluation semantics rules. The only people who should have to know what the abstract syntax looks like are the people writing the interpreter.

Now, let's write down a specification of the dynamic behaviour of this language, using the method of evaluation semantics. First, we should ask: what is our goal? How will we know when we have a complete (not necessarily *good*, but complete) specification? One answer (which is not always the right answer, but will work just fine for this language) is: if we can specify the meaning of all expressions that are syntactically well-formed (according to the EBNF for $\langle \texttt{AE} \rangle$), then we have a complete specification.

Thus, we need to specify the meaning of each of the three syntactic cases ($\langle \texttt{num} \rangle$, $+$ and $-$) in the EBNF. Since we're going to use evaluation semantics, we need to specify what a $\langle \texttt{num} \rangle$ evaluates to, what a $+$ evaluates to, and what a $-$ evaluates to.

This language is so tiny that there's only one reasonable way it can work: $+$ should add, and $-$ should subtract. So we can focus on *how* to write down an evaluation semantics, rather than spend time wondering if we're making good design decisions.

We want to be *really* precise, so let's try to be a little more precise than just saying "$+$ should add, and $-$ should subtract". Just as CPSC 110 shows how to follow a data definition (for example, if you need to write a function that takes a BST (binary search tree), you need to write a case for 'false' and a case for '(make-node ...)'), let's try to follow the BNF:

- we need to say what a number evaluates to,
- we need to say what a $+$ evaluates to, and

- we need to say what a $-$ evaluates to.

This is a little vague, though, because we didn't mention the subexpressions of $+$ and $-$. Let's fix that, and also (in the first case) mention the specific number!

- we need to say what a number n evaluates to,
- we need to say what $\{+\ ae1\ ae2\}$ evaluates to, and
- we need to say what $\{-\ ae1\ ae2\}$ evaluates to.

Here, $ae1$ stands for the first subexpression, and $ae2$ stands for the second subexpression.

Now let's actually say (in English) what these things evaluate to. A number shouldn't *do* anything, so we'll say that it evaluates to itself:

- A number n evaluates to n.

What should $\{+\ ae1\ ae2\}$ evaluate to? Well, that depends on what $ae1$ and $ae2$ are. Or rather, what they evaluate to. So let's start there.

- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to ...

We want $+$ to *add*, so it needs to add n_1 to n_2.

- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to $n_1 + n_2$.

Now we can give meaning to $-$ in the same way, resulting in something reasonably precise (it's still in English):

- A number n evaluates to n.
- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to $n_1 + n_2$.
- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{-\ ae1\ ae2\}$ evaluates to $n_1 - n_2$.

3.4.1 Rules

We're now very close to an evaluation semantics! In fact, all we have to do is rewrite the above using some funny notation: Instead of "ae evaluates to n", we'll write "$ae \Downarrow n$". And instead of "If... then ...", we'll use a horizontal line, like this:

$$\frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{\{-\ ae1\ ae2\} \Downarrow n_1 - n_2}\ \text{Eval-sub}$$

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \qquad \ldots \qquad \text{premise}_m}{\text{conclusion}}\ \text{rule name}$$

The part below the line is called the *conclusion*, and the parts above the line are called the *premises*. To the right of the line, we often write the name of the rule. We can read a rule as follows: To derive the conclusion, we must satisfy each of the premises. In other words, if the premises are satisfied, we have shown the conclusion. Or, very briefly, "if premises, then conclusion".

This notation was invented in the 1930s by the logician Gerhard Gentzen, for the purpose of formally expressing mathematical proofs. At the moment, we're using this notation to talk about what evaluation means in a (very small) programming language, but the notation is much more general. The logical statements that appear as conclusions are called *judgments*.

3.4.1.1 When is a judgment derivable?

A judgment is *derivable* if we can get it by applying rules. You may want to think of "derivable" as meaning "true"; this is okay, as long as you keep in mind that "truth" is entirely a matter of what rules you have.

3.4.1.2 Premises and conclusions

Rules always have a conclusion, but they don't have to have premises. In fact, to write down the rule for our first case ("A number n evaluates to n"), we don't need any premises:

$$\frac{}{n \Downarrow n}\ \text{Eval-num}$$

Often, the evaluation rule for a syntactic form will have exactly one premise for each smaller expression it contains. A number doesn't contain any subexpressions, so the evaluation rule for numbers doesn't have any premises. On the other hand, {− ae1 ae2} has two subexpressions, so its rule has two premises.

3.4.1.3 Applying rules

What can you do with a rule? You can *apply* it, by filling in its "meta-variables". Here, our "meta-variables" are n (in Eval-num), and ae1, ae2, n_1, and n_2 in Eval-add and Eval-sub. A meta-variable is a placeholder: we can fill in ae1 and ae2 with ⟨AE⟩'s, and we can fill in n, n_1 and n_2 with numbers.

This is easier to see with an example. Given the rule

$$\frac{}{n \Downarrow n}\ \text{Eval-num}$$

we can apply it by plugging in an actual number for the meta-variable n:

$$\frac{}{7 \Downarrow 7}$$

Once we've applied Eval-num, we have an *evaluation derivation* of $7 \Downarrow 7$, and say that we have *derived* $7 \Downarrow 7$.

Note that, unlike our EBNF grammar—where we wrote ⟨AE⟩ twice in the production for + to refer to (possibly) *different* expressions—writing n twice in the rule Eval-num means that we have to substitute the same number.

We can similarly derive $6 \Downarrow 6$:

$$\frac{}{6 \Downarrow 6}$$

This gives us two derivations, one of $7 \Downarrow 7$ and one of $6 \Downarrow 6$, so we have enough derivations to apply Eval-sub:

$$\frac{\dfrac{}{7 \Downarrow 7} \qquad \dfrac{}{6 \Downarrow 6}}{\{- \ 7 \ 6\} \Downarrow 1}$$

We got this by looking at the rule Eval-sub, plugging in 7 for ae1, plugging in 6 for ae2, plugging in 7 for n_1, and 6 for n_2. The conclusion of Eval-sub says "$\ldots \Downarrow n_1 - n_2$", which—after plugging in for n_1 and n_2—is $\ldots \Downarrow 7 - 6$, which is $\ldots \Downarrow 1$.

Notice that this derivation of $\{- \ 7 \ 6\} \Downarrow 1$ looks like a tree (oriented the natural way, with the root at the bottom, rather than the usual computer science way). And in fact, derivations are also called derivation trees. This is a nice feature of Gentzen's notation: derivations "fit together" visually.

Here's a slightly larger example:

$$\dfrac{\dfrac{\dfrac{}{20 \Downarrow 20} \quad \dfrac{}{2 \Downarrow 2}}{\{+ \ 20 \ 2\} \Downarrow 22} \quad \dfrac{\dfrac{}{7 \Downarrow 7} \quad \dfrac{}{6 \Downarrow 6}}{\{- \ 7 \ 6\} \Downarrow 1}}{\{- \ \{+ \ 20 \ 2\} \ \{- \ 7 \ 6\}\} \Downarrow 21}$$

It's often useful to write the names of the rules being applied (later languages will have more than just three rules!):

$$\dfrac{\dfrac{\dfrac{}{20 \Downarrow 20}\,\text{Eval-num} \quad \dfrac{}{2 \Downarrow 2}\,\text{Eval-num}}{\{+ \ 20 \ 2\} \Downarrow 22}\,\text{Eval-add} \quad \dfrac{\dfrac{}{7 \Downarrow 7}\,\text{Eval-num} \quad \dfrac{}{6 \Downarrow 6}\,\text{Eval-num}}{\{- \ 7 \ 6\} \Downarrow 1}\,\text{Eval-sub}}{\{- \ \{+ \ 20 \ 2\} \ \{- \ 7 \ 6\}\} \Downarrow 21}\,\text{Eval-sub}$$

3.4.2 Evaluation rules for AEs

In PL research papers, it's customary to collect all the evaluation rules together, and throw one giant figure at the reader. Fortunately, we only have three rules.

$$\dfrac{}{n \Downarrow n}\,\text{Eval-num} \qquad \dfrac{ae1 \Downarrow n_1 \quad ae2 \Downarrow n_2}{\{+ \ ae1 \ ae2\} \Downarrow n_1 + n_2}\,\text{Eval-add} \qquad \dfrac{ae1 \Downarrow n_1 \quad ae2 \Downarrow n_2}{\{- \ ae1 \ ae2\} \Downarrow n_1 - n_2}\,\text{Eval-sub}$$

The section "When is a judgment derivable?", above, may help with the following exercises.

Exercise 2. Using the above rules, is $\{* \ 1 \ 2\} \Downarrow 3$ derivable?
How about $\{* \ 1 \ 2\} \Downarrow 2$?

Exercise 3. Suppose we added a rule

$$\dfrac{ae1 \Downarrow n_1 \quad ae2 \Downarrow n_2}{\{* \ ae1 \ ae2\} \Downarrow 2}\,\text{Eval-mult}$$

Now, using the 3 rules above *and* Eval-mult, is $\{* \ 1 \ 2\} \Downarrow 2$ derivable?

3.5 From the rules to an interpreter

Now we'll write an interpreter that follows our evaluation rules. This interpreter will turn out to do the same thing as PLAI's interpreter from Chapter 2. (**2016 note:** Don't worry if you haven't read that chapter.) The difference is how we got there. Once you understand how to write interpreters based on evaluation rules, you can take evaluation rules you've never seen before—and that may define a language with features you've never heard of—and write an interpreter that follows those rules.

You won't get that skill instantly just from this one tiny language, but you have to start somewhere! And it's easier to start here than with something like *The Definition of Standard ML*.

3.5.1 Restating the rules in abstract syntax

It's easier to work with abstract syntax—the "AE" defined with define-type—than concrete syntax, so our interpreter will accept programs in abstract syntax. You can learn to mentally translate between concrete and abstract syntax, but for now, let's explicitly translate the rules to abstract syntax. We just have to change all the AEs, inserting the constructors Num, Add and Sub.

$$\frac{}{(\mathsf{Num}\ n) \Downarrow n}\ \text{Eval-num} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\mathsf{Add}\ ae1\ ae2) \Downarrow n_1 + n_2}\ \text{Eval-add} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\mathsf{Sub}\ ae1\ ae2) \Downarrow n_1 - n_2}\ \text{Eval-sub}$$

This shows something interesting, though: the animals on each side of the "evaluates to" arrow ($\Downarrow$) are not the same kind of animal.[3] In Eval-num, we have an AE, $(\mathsf{Num}\ n)$, on the left of $\Downarrow$, but a plain number n on the right. In the concrete syntax, we didn't write Num explicitly, so we couldn't see this difference. We could have chosen, instead, to "evaluate cats to cats" and produce an AE on the right, but it's a little more convenient to produce a number. (Later in 311, we'll define other flavours of operational semantics that don't work this way.)

The job of writing an interpreter for AEs boils down to writing a function that answers this question:

"Given an ae, find a number n such that $ae \Downarrow n$."

During lecture, we wrote the following function:

```
(define (interp ae)
  (type-case AE ae
    [Num (n) n]
    [Add (ae1 ae2)
        (let ([n1 (interp ae1)]
              [n2 (interp ae2)])
          (+ n1 n2))]
    [Sub (ae1 ae2)
        (let ([n1 (interp ae1)]
              [n2 (interp ae2)])
          (- n1 n2))]
    ))
```

Our `interp` function behaves the same as the `calc` function in PLAI, but our function has more let-bindings. This is more verbose, but strengthens the connection between our interpreter and the rules. For example, the expression `(+ n1 n2)` is a direct Racket translation (parentheses and a prefix operator +) of the $n_1 + n_2$ that appears in the conclusion of Eval-add.

[3] In honour of my undergrad discrete math professor's advice: "You must always ask yourself: what kind of an animal is it?"

4 Evaluation semantics: Functions

Updated 2016-09-25 with substitution (Figure 4.2) and some discussion from the 2016-09-23 lecture (Section 4.4.3)

4.1 Topics discussed

- Recipe for extending a language
- The "Fun language": concrete syntax, abstract syntax
- The Fun language: evaluation rules
  - why evaluation in Fun produces a *value*, not just a number
  - evaluation rule for `lam`
  - updated evaluation rules for features already in our language
  - rule for function application?
- rule for function application: Eval-app-expr
- the "expression strategy"
- example with Eval-app-expr
- the "method of hope"; complete derivations
- another example with Eval-app-expr
- the "value strategy" and Eval-app-value
- advantages and disadvantages of each strategy
- interpreter for Fun (**Racket code**: `dynsem-fun.rkt` ⇐ **2015 VERSION, OUT OF DATE**)
- first-class functions
  - mathematicians also think they're weird
  - functions are values...
  - ...but are displayed in an unhelpful way by most language implementations
  - when are functions equal?

4.2 Functions

We started our journey through evaluation semantics with a language of arithmetic expressions. We got a slightly larger language by adding the Let construct, which lets us give names to values and then use those names. To model Let in the evaluation semantics, we had to define substitution.

Adding just one more feature will get us to a surprisingly powerful language.

To add functions, we'll try to follow the same recipe as for Let:

1. **Extend the concrete syntax** (EBNF grammar).
2. **Extend the abstract syntax** (**define-type**).
3. **Add evaluation rules.**

After finishing these steps, we will have extended our *language*: a language is defined by its syntax and semantics. To implement the language, we'll need to extend our parsing function (to reflect the new syntax) and our interpreter (to reflect the new evaluation rules).

Remark. In real life, or at least in real programming languages research, there would probably be a fourth step: **Prove that the new evaluation rules have good properties.** Exactly what properties are "good" depends on the language. For our language so far, one good property (the only one I can think of right now) would be *determinism*, which says that evaluating the same expression should give consistent results:

"If $e \Downarrow n_1$ and $e \Downarrow n_2$, then $n_1 = n_2$."

Later in 311, we'll discuss these sorts of properties (without actually proving them).

4.3 The Fun language: syntax

It's time to add functions to our language. Nearly a century of tradition would have us use the syntax "lambda" or λ, but to help distinguish the lambda in Fun from the lambda in Racket (which can also be written λ), we'll use "Lam".

Instances of the identifier bound by `Lam` can be represented by `Id`, just as we did for `Let`.

⟨E⟩ ::= ⟨num⟩
| {+ ⟨E⟩ ⟨E⟩}
| {− ⟨E⟩ ⟨E⟩}
| ⟨symbol⟩
| {Let ⟨symbol⟩ ⟨E⟩ ⟨E⟩}
| {Lam ⟨symbol⟩ ⟨E⟩}

And here's the abstract syntax:

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Id (name symbol?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Lam (name symbol?) (body E?)]
)
```

We're not done with the syntax, though. In a previous lecture, I mentioned that a language can be organized by what kinds of data it has, and how it "introduces" and "eliminates" each kind of data. Until now, our languages only had one kind of data, numbers; functions are a new kind of data. We can introduce functions with "Lam", but we need a way to use them. We'll call this "App". The first expression will represent the function being applied, and the second expression will represent the argument—what the function is being applied to.

⟨E⟩ ::= ⟨num⟩
| {+ ⟨E⟩ ⟨E⟩}
| {− ⟨E⟩ ⟨E⟩}
| ⟨symbol⟩
| {Let ⟨symbol⟩ ⟨E⟩ ⟨E⟩}
| {Lam ⟨symbol⟩ ⟨E⟩}
| {App ⟨E⟩ ⟨E⟩}

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
)
```

4.4 The Fun language: Evaluation rules

We'll try to reuse all the rules from before. Because we don't know yet whether that will work, we'll write a ? before each rule name to emphasize its provisional status.

When we write a meta-variable e in a rule, it will stand for a Fun expression.

$$\frac{}{(\mathsf{Num}\ n) \Downarrow n}\ \text{?Eval-num} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\mathsf{Add}\ e1\ e2) \Downarrow n_1 + n_2}\ \text{?Eval-add} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\mathsf{Sub}\ e1\ e2) \Downarrow n_1 - n_2}\ \text{?Eval-sub}$$

$$\frac{e1 \Downarrow n1 \qquad [n1/x]e2 \Downarrow n2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow n2}\ \text{?Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \text{free-variable-error}}\ \text{?Eval-free-identifier}$$

We added two productions to the EBNF and two variants to the abstract syntax, so we expect to need two new evaluation rules. We might also expect (following the pattern of Eval-add, Eval-sub, and Eval-let) that, since a Lam has one expression inside it and an App has two expressions inside it, the rule for Eval-lam will have one premise and the rule for Eval-app will have two premises:

$$\frac{??}{(\mathsf{Lam}\ x\ e1) \Downarrow ??}\ \text{??Eval-lam} \qquad \frac{?? \qquad ??}{(\mathsf{App}\ e1\ e2) \Downarrow ??}\ \text{??Eval-app}$$

Let's think about ??Eval-lam. What should its premise be? The only expression we have is $e1$, so maybe we should evaluate $e1$ in the premise.

$$\frac{e1 \Downarrow n1}{(\mathsf{Lam}\ x\ e1) \Downarrow n1??}\ \text{??Eval-lam}$$

This seems to follow the pattern of our other rules, but (as with programming) we should think about examples (test cases). What is the *simplest possible* function? I claim it is the identity function, $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$. So let's try that function with our proposed rule:

$$\frac{(\mathsf{Id}\ x) \Downarrow \_\_\_}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow \_\_\_}$$

Now we have a problem: the expression $(\mathsf{Id}\ x)$ doesn't evaluate to anything—instead, we get a "free-variable-error". So we can't derive $(\mathsf{Id}\ x) \Downarrow \_\_\_$.

The problem is that we're trying to evaluate what's *inside* the function before we've applied it *to* anything! We don't know what x is until we pass the function an argument. A function is a machine that turns arguments into results; evaluating a function without an argument is like running a dishwasher when it's empty.

We already have numbers that evaluate to themselves (Eval-num), so we'll make functions evaluate to themselves as well.

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam}$$

Irritatingly, this doesn't quite match what we've done so far. Instead of always evaluating to numbers—deriving

$$\ldots \Downarrow n$$

where n is a number, we can now use Eval-lam to derive

$$\ldots \Downarrow (\mathsf{Lam}\ x\ e1)$$

We could say that evaluation produces either a number n or an expression of the form (Lam x e1). It will be a little easier to say that evaluation produces a *value*, where a value is a particular kind of expression: one that is either (Num n), or (Lam x e1).[1]

Making this change requires several changes to our rules:

$$\frac{}{(\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Add } e1\ e2) \Downarrow (\text{Num } (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Sub } e1\ e2) \Downarrow (\text{Num } (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{Let } x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\text{Id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\text{Lam } x\ e1) \Downarrow (\text{Lam } x\ e1)}\ \text{Eval-lam}$$

Interestingly, one of the rules—Eval-let—became simpler, and more general: it should now work with any kind of value $v1$, not just numbers.

Figure 4.1 summarizes all our rules so far—we're still missing a rule for App.

$$\frac{}{(\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Add } e1\ e2) \Downarrow (\text{Num } (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Sub } e1\ e2) \Downarrow (\text{Num } (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{Let } x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\text{Id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\text{Lam } x\ e1) \Downarrow (\text{Lam } x\ e1)}\ \text{Eval-lam}$$

Figure 4.1 Evaluation rules for Fun (still missing a rule for App)

Figure 4.2 defines substitution for this language. The rules for substitution into a Lam echo the rules for substitution into a Let, except that a Lam has only one expression inside it.

4.4.1 Evaluating application

Here's our guess at the shape of the Eval-app rule, with a lot of ??'s.

$$\frac{?? \qquad ??}{(\text{App } e1\ e2) \Downarrow ??}\ ??\text{Eval-app}$$

Ideas?

At this point, I was expecting to (eventually) reach a particular rule, but it's not the one that you suggested during lecture. (I should have expected this, since I suggested that we have a rule with two premises, and the one I was thinking of has three premises!) However, the rule you suggested is quite reasonable.

The first suggestion was to evaluate e1 to a Lam. Why does it have to be a Lam? Well, evaluation produces a value. We have two kinds of values so far: numbers and Lams. Applying a number as a

[1] By themselves, values are inert. They don't do anything. A function does nothing until it's called. Unfortunately, many "real" programming languages make this hard to remember. For example, DrRacket claims that the result of evaluating (**lambda** (x) x) is "#<procedure>". But you should think of that as just a strange notation for the function you entered. It's still (**lambda** (x) x).

$$
\begin{array}{rcll}
[v/x](\mathsf{Num}\ n) & = & (\mathsf{Num}\ n) & \\
[v/x](\mathsf{Add}\ e1\ e2) & = & (\mathsf{Add}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{Sub}\ e1\ e2) & = & (\mathsf{Sub}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{App}\ e1\ e2) & = & (\mathsf{App}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{Id}\ y) & = & v & \text{if } x = y \\
[v/x](\mathsf{Id}\ y) & = & (\mathsf{Id}\ y) & \text{if } x \neq y \\
[v/x](\mathsf{Let}\ y\ e1\ e2) & = & (\mathsf{Let}\ y\ [v/x]e1\ e2) & \text{if } x = y \\
[v/x](\mathsf{Let}\ y\ e1\ e2) & = & (\mathsf{Let}\ y\ [v/x]e1\ [v/x]e2) & \text{if } x \neq y \\
[v/x](\mathsf{Lam}\ y\ eB) & = & (\mathsf{Lam}\ y\ eB) & \text{if } x = y \\
[v/x](\mathsf{Lam}\ y\ eB) & = & (\mathsf{Lam}\ y\ [v/x]eB) & \text{if } x \neq y
\end{array}
$$

Figure 4.2 Substitution for Fun

function doesn't make much sense, so it's got to be a Lam. Also, requiring $e1$ to evaluate to a Lam corresponds to Eval-add, where the expressions being added have to evaluate to Nums.

$$\frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad ??}{(\mathsf{App}\ e1\ e2) \Downarrow ??}\ \text{?Eval-app}$$

I can't think of any other way of writing this first premise. This is good progress, so I dropped one of the ?'s from the name of the rule.

Now we come to a "fork in the road". The choice we make now will decide what *evaluation strategy* this language has. This is usually considered an important and fundamental language design choice, especially for functional languages (like Racket and this Fun language).

4.4.1.1 The "expression strategy"

The next suggestion during lecture was to substitute $e2$ for x, like this:[2]

$$\frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad [e2/x]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-expr}$$

We'll call this the *expression strategy*[3], because we're taking the expression $e2$—the argument being passed to $(\mathsf{Lam}\ x\ eB)$—and substituting it for x, *without* evaluating $e2$.

Let's look at some examples.

- **Evaluating an application of the identity function.**

  Suppose we apply the identity function $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$ to some simple expression, like $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$.

  (It would be even simpler to apply the identity function to a Num, but that wouldn't illustrate what I'm trying to illustrate.)

[2] During lecture, I called the result of evaluation $v2$ rather than v. It will be less confusing to call it v.
[3] There is a more traditional name, which we'll talk about in due course.

So we need to derive

$$(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow \cdots$$

First we "match" expressions to meta-variables in Eval-app-expr:

$$(\mathsf{App}\ \underbrace{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}_{e1}\ \underbrace{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))}_{e2}) \Downarrow \cdots$$

Then we plug in those expressions:

$$\frac{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ eB) \qquad [(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x]eB \Downarrow v}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow v}\ \text{Eval-app-expr}$$

We need to be careful about Gentzen's notation. We *want* to apply the rule Eval-app-expr, but we haven't really applied it yet, because we haven't derived its two premises. A rule is an "if. . . then . . . " statement; you don't get to conclude the "then" part without showing that the "if" holds. What I wrote above is based on the power of hope: I *want* to derive $(\mathsf{App}\ \cdots\ \cdots) \Downarrow v$, so I'm going to *try* to apply Eval-app-expr. To apply Eval-app-expr, I need to derive each premise, using the same method of hope.

At each step in this process, any leaf in my derivation tree that doesn't have a horizontal line over it is a *goal* that remains to be proved.[4]

If I can get a derivation tree whose leaves all have horizontal lines over them, I will know that I have derived $(\mathsf{App}\ \cdots\ \cdots) \Downarrow v$, but not before.

We now want to derive

$$(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ eB) \qquad \text{(first goal)}$$

and

$$[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x]eB \Downarrow v \qquad \text{(second goal)}$$

For the first evaluation, we have a rule that evaluates Lams: Eval-lam. Plugging in for the meta-variable $e1$ in that rule, we get

$$\frac{}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ \underbrace{(\mathsf{Id}\ x)}_{eB})}$$

Note that this tells us what eB is, so we can revise our second goal by plugging in $(\mathsf{Id}\ x)$ for eB:

$$[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x]\ (\mathsf{Id}\ x) \Downarrow v \qquad \text{(second goal, revised)}$$

Since *subst* is now applied to "real" arguments (without unknown meta-variables), we can use the definition of *subst*. We really should revise the definition of *subst* to our extended language, but the old version works for this example: replacing all instances of the identifier x in $(\mathsf{Id}\ x)$ with $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$ is just $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$.

$$(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow v \qquad \text{(second goal, revised again)}$$

[4] If you have used a logic programming language, such as Prolog (taught in CPSC 312), yes, there is a connection; we may not have time to explore it in 311, though.

Exercise 4. Derive this second goal, following the "method of hope" as above. (This is *not* exactly like a similar example for the AE language, because we changed our notion of evaluation to produce expressions—more specifically, values—rather than numbers.) I left some space above for you to write the derivation tree.

I'll assume you've derived the second goal, and to keep track, I'll put a checkmark ✓ above it. I'm also assuming you got (Num 5), so I'm plugging that in for the meta-variable v.

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-lam} \qquad \overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)}}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow (\mathsf{Num}\ 5)}\ \text{Eval-app-expr}$$

Everything in this tree has either a horizontal line or a checkmark, so we have a complete evaluation derivation.

Remember: Following the method of hope, a derivation tree is **complete** if each leaf of the tree is either (1) an application of a rule with no premises (for example, Eval-num), or (2) a conclusion of some other complete derivation tree.

You can tell (1) because the leaf has a horizontal line with nothing above it.

To keep track of (2), write a checkmark above the leaf.

If a leaf doesn't have a horizontal line or a checkmark, that leaf is a goal that needs to be be derived, and the derivation is **incomplete**.

- **Evaluating an application of the doubling function.**

  In Fun, we can write a function that doubles its argument:

  $$(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))$$

  What happens when we apply this function to (Add (Num 2) (Num 3))? Hopefully, we will get 10, since $2\cdot(2+3) = 10$. But remember that we are evaluating to expressions now, so we actually want to evaluate to (Num 10).

$$\dfrac{\dfrac{}{\ldots \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))}\ \text{Eval-lam} \qquad [(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x](\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)) \Downarrow v}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow v}\ \text{Eval-app-expr}$$

  To save space, I wrote "…", but since Eval-lam has the same thing on both sides of the "⇓", it has to be $(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))$.

  The second premise of Eval-app-expr has neither a horizontal line above nor a checkmark, so we have an incomplete derivation. To figure out which rule to try, we need to know what expression we have, so we look up the definition of *subst*. That gives:

$$\dfrac{\dfrac{}{\ldots \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))}\ \text{Eval-lam} \qquad (\mathsf{Add}\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow v}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow v}\ \text{Eval-app-expr}$$

We now know exactly what expression we have, so we can try to apply a rule. We have an Add, so we'll try Eval-add.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))}\text{Eval-lam} \quad \dfrac{(\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ \_\_\_) \qquad (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ \_\_\_)}{(\text{Add}\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ \_\_\_+\_\_\_)}\text{Eval-add}}{(\text{App}\ (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ \_\_\_+\_\_\_)}\text{Eval-app-expr}$$

Conveniently, you already did a complete derivation for (Add (Num 2) (Num 3)) and found that this expression evaluated to (Num 5), so we can fill in all of the blanks with 5.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))}\text{Eval-lam} \quad \dfrac{(\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ 5)\ \checkmark \qquad (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ 5)\ \checkmark}{(\text{Add}\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ 5+5)}\text{Eval-add}}{(\text{App}\ (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ 5+5)}\text{Eval-app-expr}$$

Finally, we replace $5 + 5$ with 10 in the derivation tree.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))}\text{Eval-lam} \quad \dfrac{(\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ 5)\ \checkmark \qquad (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow (\text{Num}\ 5)\ \checkmark}{(\text{Add}\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ 10)}\text{Eval-add}}{(\text{App}\ (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ 10)}\text{Eval-app-expr}$$

Notice that in last example, we ended up with *two* copies of the same evaluation derivation (the one you did as an exercise). This is because we had two instances of x in the body of the function being applied, and our rule Eval-app-expr substitutes the entire expression (Add (Num 2) (Num 3)).

Does that really matter, except for making the derivation tree bigger? Well, maybe. If the evaluation semantics seems to be doing work twice, an interpreter based on the semantics will *also* do that work twice! That doesn't matter much for this small example, but imagine if, instead of the argument being (Add (Num 2) (Num 3)), it were some expression that computed the sum of the first million prime numbers. Our interpreter would evaluate that huge expression twice, even though it only appears once in the input expression (App $\cdots$ $\cdots$).

4.4.2 The "value strategy"

An alternative strategy, which we'll call the *value strategy*, is to have this rule instead of Eval-app-expr:

$$\dfrac{e1 \Downarrow (\text{Lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\text{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

The first premise is the same as Eval-app-expr, but a new second premise evaluates *e2 immediately*, and in a third premise, we substitute the *result* of that evaluation for x.

We can see the difference from Eval-app-expr by evaluating the same Fun expression we evaluated just above, the doubling function applied to (Add (Num 2) (Num 3)). The expression still evaluates to (Num 10), but the derivation looks rather different:

$$\dfrac{\dfrac{}{\ldots \Downarrow (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))}\text{Eval-lam} \quad (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3)) \Downarrow \overbrace{(\text{Num}\ 5)}^{v2}\ \checkmark \qquad (\text{Add}\ (\text{Num}\ 5)\ (\text{Num}\ 5)) \Downarrow (\text{Num}\ 10)}{(\text{App}\ (\text{Lam}\ x\ (\text{Add}\ (\text{Id}\ x)\ (\text{Id}\ x)))\ (\text{Add}\ (\text{Num}\ 2)\ (\text{Num}\ 3))) \Downarrow (\text{Num}\ 10)}\ \text{Eval-app-value}$$

4.4.3 Other possible rules

Based on a suggestion during lecture (in 2016), another possible rule would be

$$\frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad e2 \Downarrow v2}{(\mathsf{App}\ e1\ e2) \Downarrow [v2/x]eB}\ \text{Eval-app-alternate}$$

In this rule, the substitution $[v2/x]$ happens in the conclusion, not in a premise.

If the body eB of the Lam being applied is simple enough, this rule works in the same way as Eval-app-value. But for a function like $\big(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big)$, we would derive

$$\frac{e1 \Downarrow \big(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\big) \qquad (\mathsf{Num}\ 5) \Downarrow (\mathsf{Num}\ 5)}{\big(\mathsf{App}\ \underbrace{\big(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big)}_{e1}\ \underbrace{(\mathsf{Num}\ 5)}_{e2}\big) \Downarrow \underbrace{(\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))}_{[v2/x]eB}}\ \text{Eval-app-alternate}$$

(I left out the details of evaluating the premises.) Here the result of evaluation is $[v2/x]eB = \big[(\mathsf{Num}\ 5)/x\big](\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)) = (\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))$, which is neither a Num nor a Lam; to get the expected answer, (Num 10), we would need to evaluate (Add (Num 5) (Num 5)). The rule gets us closer to the answer, but not all the way, and evaluation semantics is supposed to give us a final answer.

It's possible to design a system of rules that takes smaller steps from an expression to a value. In that kind of semantics, called a "small-step" semantics, we might say that

$$\big(\mathsf{App}\ \big(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big)\ (\mathsf{Num}\ 5)\big) \longrightarrow (\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))$$

and that

$$(\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5)) \longrightarrow (\mathsf{Num}\ 5)$$

Such a semantics has several nice features, including being "closer to the machine" (in fact, assembly language can be modelled using a small-step semantics) and allowing us to specify the behaviour of programs that don't terminate. We will use a small-step semantics for some later parts of 311. However, the rules of evaluation semantics (or "big-step semantics") are easier to implement. Since the first part of 311 emphasizes the connection between rules, which specify language behaviour, and interpreters that implement that behaviour, we are using the easier-to-implement style—evaluation semantics—rather than small-step semantics.

4.4.4 Advantages and disadvantages

Think about the "expression strategy" and the "value strategy". The value strategy gave us a smaller derivation for one of our examples—and would also give us a faster interpreter for that example.

- Both strategies seem to be giving us the same answers—evaluation is giving us the same values. Is that true in general?
  - There are different ways to read "in general". For our language[5], we will indeed get the same value regardless of which strategy we use, for any expression. (Caveat: I haven't proved this.) The size of the derivations, and the amount of time the interpreter will take, may be very different, but we'll get the same value (either a Num or a Lam).

[5] Or rather, our languages: a language is syntax and semantics together, so we should really talk about two languages: the "Fun with Eval-app-expr" language, and the "Fun with Eval-app-value" language.

  - If we mean languages generally, there are languages where this doesn't hold. In a language with *effects*, the argument *e*2 might do something that allows us to distinguish the two evaluation strategies. For example, *e*2 might print a string, and a different number of strings would be printed depending on the evaluation strategy. The values returned wouldn't change, however: either you print once, and return a value, or you print twice and return the same value.

    We *could* get different values, however, if our language had *mutable state*, such as an incrementable counter. If *e*2 increments this counter, the contents of the counter might affect the value returned.

    Languages with mutable state have more complicated semantics, which we'll look at later on.

- Are there any Fun expressions where the *expression* strategy (Eval-app-expr) would be faster?

  - Yes—expressions that apply a function that doesn't use its argument:

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Num}\ 4)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))}\ \text{Eval-lam} \qquad [(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x](\mathsf{Num}\ 4) \Downarrow v}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow v}\ \text{Eval-app-expr}$$

    Since x doesn't occur in (Num 4), the result of *subst* on (Num 4) is just (Num 4):

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Num}\ 4)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))}\ \text{Eval-lam} \qquad (\mathsf{Num}\ 4) \Downarrow v}{(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))\ \underbrace{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))}_{e2}) \Downarrow v}\ \text{Eval-app-expr}$$

    Here, we never evaluate the argument *e*2 at all! The value strategy would evaluate *e*2 even though it's not needed.
  - There's another evaluation strategy, *lazy evaluation*, which doesn't evaluate the argument *e*2 until it's used inside the Lam. At that time, it evaluates *e*2 and remembers the value it gets. Other instances of x will reuse that value instead of evaluating *e*2 again. This strategy is a little more complicated to define, but we'll come back to it later in 311.

- Does evaluation in Racket work like the expression strategy, or like the value strategy? How about Java? Haskell? Algol-60?

  - What Racket does isn't exactly the same as either of our evaluation rules, but it's very close to the value strategy.
  - Java: same answer. (In Java, and similar languages, you often pass *pointers* or *references* around, but those are really just values, albeit of a different kind than the values in Fun.)
  - Haskell uses lazy evaluation (see above), so it's kind of like the expression strategy.
  - Algol-60 supports *both* the value strategy and the expression strategy. The expression strategy is the default, but programmers can designate specific function arguments as following the value strategy. (The Algol-60 committee had *just invented the expression strategy*. Years later, several committee members were still angry that the report's editor, Peter Naur—also the 'N' in 'BNF'—decided, on his own, to make the expression strategy be the default.)

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

Figure 4.3 Evaluation rules for Fun

4.5 From the Fun rules to a Fun interpreter

Earlier, we said that an interpreter should do this:

"Given an e, find a number n such that $e \Downarrow n$."

For Fun, we have a more general idea of the result of evaluation that includes functions as well as numbers, and we said that functions (Lam $\cdots$) and numbers (Num $\cdots$) are collectively *values*, so a Fun interpreter should do this instead:

"Given an e, find a value v such that $e \Downarrow v$."

(Live-coding time...)

4.6 Collected rules for Fun

Figure 4.3 collects all the rules, showing Eval-app-value rather than Eval-app-expr.

4.7 Fly first-class, for free

If we wrote our interpreter correctly, we now have a programming language that is quite similar to the λ-calculus (which was invented by Alonzo Church in the 1930s, and extensively studied ever since). As a programming language, the λ-calculus has very few features (it doesn't even do arithmetic... at least not in a way that you'd recognize), but it does have functions that are *first-class*—functions that can take other functions as arguments, and return functions. You may not have noticed it, but our rules have no trouble with first-class functions.

4.7.1 It's not just you

First-class functions are often considered a strange and advanced language feature. Back in 1967, Christopher Strachey (who worked on the semantics of programming languages, and also—rather curiously—set in motion a chain of events that led to C) pointed out that mathematicians rarely treated functions as "first-class" and hadn't even agreed on a notation for first-class functions; mathematicians seemed to have little grasp of how to use functions as values.

Today, the DrRacket "Beginning Student" language doesn't allow functions as arguments, and it doesn't allow a function to return a function. The "Intermediate Student" language allows functions

as arguments, but not as results. The "Intermediate Student with Lambda" language adds the ability to return a function (by returning a Lambda).

This distinction may be useful when learning how to program; I'm not sure it's useful when learning how to think about defining programming languages. **Functions are values;** until they are applied, they don't do anything, just as numbers don't do anything until you do arithmetic on them.

Unfortunately, most "real" programming languages make that hard to remember. For example, DrRacket claims that the result of evaluating (**lambda** (x)x) is "#<procedure>". But that's a bad, secretive notation for the function you entered; you should think of it as being (**lambda** (x)x). Two other functional languages, SML and OCaml, also refuse to show you the inside of a function.

```
Standard ML of New Jersey v110.72 [built: Tue Jan 11 13:30:58 2011]
- 2 + 2;
val it = 4 : int
- (fn x => x + x) 2;
val it = 4 : int
- (fn x => x + x);
val it = fn : int -> int
```

```
        OCaml version 4.00.1

# 2 + 2;;
- : int = 4
# (fun x -> x + x) 2;;
- : int = 4
# (fun x -> x + x);;
- : int -> int = <fun>
```

Two different Haskell implementations, hugs and ghci, not only won't show it, but print mysterious error messages, just to taunt you.

```
__   __ __  __  ____   ___      ------------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ------------------------------------------

Hugs> 2 + 2
4
Hugs> (\x -> x + x) 2
4
Hugs> (\x -> x + x)
ERROR - Cannot find "show" function for:
*** Expression : \x -> x + x
*** Of type    : Integer -> Integer
```

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
```

```
Prelude> 2 + 2
4
Prelude> (\x -> x + x) 2
4
Prelude> (\x -> x + x)

<interactive>:4:1:
    No instance for (Show (a0 -> a0)) arising from a use of ~print~
    In a stmt of an interactive GHCi command: print it
Prelude>
```

Python displays the function along with its address in memory:

```
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
>>> 2 + 2
4
>>> (lambda x: x + x) (2)
4
>>> (lambda x: x + x)
<function <lambda> at 0x1023d3938>
```

Why is this? I'm not sure. It kind of makes sense for a compiler to throw away abstract syntax, but Hugs isn't even a compiler. The point of a REPL (read-eval-print loop; also known in DrRacket as "Interactions", and in many other languages as a "toplevel") is not to be efficient, but to allow "playing" with a language by typing in expressions and seeing what happens. It doesn't seem that difficult for something like SML to preserve the abstract syntax of code, at least code that you enter in the REPL. But I haven't implemented it myself, so there are probably issues I haven't thought of.

4.7.2 Looking behind the curtain

The distinction between first-class functions and "lesser" functions may matter, however, when we try to write *efficient* interpreters and compilers. But not worrying about efficiency is helpful now: Because our interpreter follows a (relatively) very simple evaluation semantics, you can get an idea of how first-class functions work *in general* by writing Fun expressions that evaluate to functions, and *looking at the functions*—our Fun language always shows you what's inside a Lam! Then you can take that *general* idea and use it when programming in Racket, OCaml, or Haskell.

This may not be too effective yet, because our Fun language is so small, but you'll add several features to it in the next assignment.

4.7.3 Unparsing

A parse function takes concrete syntax (for us, an S-expression) and builds abstract syntax. An "unparse" function takes the abstract syntax, and turns it back into concrete syntax.

I'm doing this now so that when you look at the Lams your Fun expressions evaluate to, you can read them more easily.

For convenience, I'm using quasiquote and unquote. (A fun (?) exercise: write a version of unparse that *doesn't* use quasiquote and unquote.) The file dynsem-fun.rkt has some useful links about these features.

4.7.4 Digression: equality of functions

When are functions equal? Not an easy question!

In Racket, after defining a function using **define**, that function is equal to itself. But if we write identical expressions using **lambda**, they are not equal.

```
> (equal? unparse unparse)
#t
> (equal? (lambda (x) x) (lambda (x) x))
#f
```

Python works similarly.

SML, rather characteristically, just refuses to let you compare functions at all:

```
- (fn x => x) = (fn x => x);
stdIn:1.1-1.26 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) * ('X -> 'X)
  in expression:
    (fn x => x) = (fn x => x)
```

Its complaint is that functions don't have "equality type"; they don't have a type for which SML defines equality. The principle here is that the answer to whether two functions are equal is most likely useless—it would be reasonable to expect that `(lambda (x) x)` would be equal to itself, but it's not, so SML just doesn't define equality on functions at all.

OCaml defines two (at least) kinds of equality: $=$ doesn't work for functions (with an exception, rather than a type error), and $==$ works like equal? in Racket: it *might* return true for functions that are the same, but it might just return false.

```
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
# (fun x -> x) == (fun x -> x);;
- : bool = false
# let identity = fun x -> x;;
val identity : 'a -> 'a = <fun>
# identity == identity;;
- : bool = true
```

Neither approach to function equality (not defining it at all, or defining an equality test that often says "no, they're not equal" even for functions with identical source code) is totally satisfying. A mathematician's answer to the question, "Are the function $f(x) = x+1$ and the function $g(y) = y+2-1$ the same?" would be (I think—I'm not really a mathematician) "yes", because both functions are *extensionally equal*: given the same arguments, they produce the same results.

In general, the question of whether two functions are extensionally equal is undecidable, and certainly difficult: imagine that the bodies of the functions f and g above were each 100,000 lines long. An interesting approach would be to implement a version of function equality that has *three* possible answers: "these functions are obviously extensionally equal", "these functions are obviously extensionally not equal", and "I don't know". I don't know if anyone has tried this approach.

4.8 Recursion

A reasonable objection to our Fun language so far is that we can't write recursive functions, so let's address that.

The approach we'll take is not to add recursive *functions* as such, but a recursion *expression* whose body can be a function. For example, we can write $(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ e))$, where e is some expression that can refer to u and x.

(It would be slightly more standard to write "fix" instead of Rec, for "fixed point", but we won't concern ourselves with whatever a fixed point might be. I mention this only to encourage you to yell at me if I write fix by accident.)

$$\begin{aligned}\langle \mathrm{E} \rangle \ ::= \ & \vdots \\ & \mid \{\mathsf{Rec}\ \langle \mathtt{symbol} \rangle\ \langle \mathrm{E} \rangle\}\end{aligned}$$

What does this thing mean? To answer (?) that, we need an evaluation rule.

$$\frac{\big[(\mathsf{Rec}\ u\ e)/u\big]e \Downarrow v}{(\mathsf{Rec}\ u\ e) \Downarrow v}\ \text{Eval-rec}$$

The identifier u in $(\mathsf{Rec}\ u\ e)$ is a way for the expression e to refer to *itself*. So, to evaluate $(\mathsf{Rec}\ u\ e)$, we replace u with... $(\mathsf{Rec}\ u\ e)$! Unfortunately, this can lead to trouble...

A very simple example of a Rec is the expression

$$\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big)$$

Evaluating this is no trouble, but the Rec doesn't really serve any purpose here: u doesn't appear in $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$, so substituting for u has no effect:

$$\frac{\big[\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big)/u\big](\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow v}\ \text{Eval-rec}$$

Using the definition of substitution, the premise is really

$$\frac{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow v}\ \text{Eval-rec}$$

Using Eval-lam, we get

$$\frac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-lam}}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-rec}$$

This is a perfectly good derivation, but we could have obtained the same value by omitting the Rec and just writing $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$.

Let's try to evaluate the simplest possible Rec expression that *does* use u.

$$\frac{\big[\big(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)\big)/u\big](\mathsf{Id}\ u) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

Rewriting our goal (the premise of Eval-rec) using the definition of *subst*, we get

$$\frac{\big(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)\big) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

So now we need to derive $(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v$. The only rule that could possibly work is Eval-rec:

$$\dfrac{\dfrac{\big[(\mathsf{Rec}\ u\ (\mathsf{Id}\ u))/u\big](\mathsf{Id}\ u) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

This new goal uses *subst*, so we follow that definition again...

$$\dfrac{\dfrac{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

This isn't going anywhere!

We should clarify our idea of what a derivation is: a derivation *must be finite*. Endlessly applying the same rule to get an infinite tree isn't allowed.

4.8.1 Base and recursive cases

Our first attempt to use Rec didn't really do anything; we wrote Rec but didn't use it, kind of like the base case of a recursive function. Our second attempt had us endlessly trying to derive the same thing (and an interpreter following the Eval-rec rule would, in fact, run forever).

A third idea:

$$(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{App}\ (\mathsf{Id}\ u)\ (\mathsf{Id}\ x)))))$$

This does use u. Evaluating this expression is fine—it evaluates to a Lam. However, when we apply that Lam to something, we'll try to evaluate forever again (though with an ever-changing goal).

Earlier, we added ifzero, so we now have a way for a Fun expression to test an argument and do different things based on it.

4.9 Soundness and completeness

What does "following the rules" really mean?

Definition 5. Completeness of the interpreter: If $e \Downarrow v$ is derivable then $(\mathsf{interp}\ e) = v$.

If completeness does not hold, we say the interpreter is *incomplete*. For example, you might forget to implement an evaluation rule.

Definition 6. Soundness of the interpreter: If $(\mathsf{interp}\ e) = v$ then $e \Downarrow v$ is derivable.

If soundness does not hold, we say the interpreter is *unsound*.

The words "is derivable" are not quite necessary, but I included them to emphasize that the definition of $e \Downarrow v$ is given by rules.

4.9.1 Bonus rant

(Skipped during lecture; feel free to skip it here too.) In "interpreter semantics", the *interpreter itself* defines what $e \Downarrow v$ means. So the definitions of soundness and completeness collapse: "If $e \Downarrow v$ then $e \Downarrow v$."

Suppose two of you are (separately) implementing interpreters. One of you implements function application in a way that corresponds to Eval-app-value, and the other implements function application in a way that corresponds to Eval-app-expr. Under interpreter semantics, you have *both* implemented a "correct" interpreter, because *the act of writing an interpreter* (according to interpreter semantics) defines what the language is.

Interpreter semantics has another drawback: you cannot construct a language definition in which behaviour is undefined, because whatever your interpreter happens to do *is* the definition of the language. Now, *which* behaviours should be left undefined can be debated, but real programming languages have multiple implementations (even if we're only counting patches and bug fixes to a single "canonical" implementation!) and run in different environments and processor architectures; you usually can't define everything.

4.9.2 Undefined behaviour

To be sound, your interpreter must not evaluate an expression successfully (that is, return a value) unless the rules say it does. So your interpreter must not return a value for the expression

$$(\mathsf{Add}\ (\mathsf{Lam}\ \cdots)\ (\mathsf{Lam}\ \cdots))$$

unless $(\mathsf{Add}\ (\mathsf{Lam}\ \cdots)\ (\mathsf{Lam}\ \cdots)) \Downarrow v$ is derivable according to the rules (which it's not for the languages Fun and Fun++ that we've discussed).

For free identifiers, we wrote a rule Eval-free-identifier that says that evaluating $(\mathsf{Id}\ x)$ is a "free-variable-error". But we haven't written rules for other "errors", like adding two Lams. Thus, your interpreter can't return a value, but it's free to treat adding two Lams any way you like. You could:

- generate an error (similar to free-variable-error);
- loop forever (which sounds kind of silly, but we already loop for (Rec u (Id u))...);
- something else entirely.

(Viktor Vafeiadis, who studies the C++ memory model, likes to give this example of undefined behaviour: "You could launch the missiles.")

Generating an error in such cases sounds like the most organized (precise) option. Writing the rules for this, however, would get rather tedious. More later.

4.9.2.1 Examples of undefined, unspecified, and implementation-dependent behaviour

- **OCaml**: Feels like one compiler (you download "ocaml", not two different compilers) but has two "back ends": one that generates machine code, and one that generates OCaml virtual machine bytecode. One back end evaluates function application left to right; the other evaluates function application right to left. If you care about order of evaluation, you need to use OCaml's let.
- **C**: Arithmetic overflow is undefined for signed integers. For unsigned integers, it must "wrap around". This seems to be because C predates the consensus that computer architectures should use "two's complement" to represent integers.[6]

[6] stackoverflow.com/questions/18195715/why-is-unsigned-integer-overflow-defined-behavior-but-signed-integer-overflow-is

- **C** (and many other languages): The size of an `int` was completely unspecified in 1970s/1980s C, and is now partly specified. C99 says that an `int` must be at least 16 bits—well, not quite. Rather, it must be able to represent values between -32767 and 32767. In two's complement representation, 16 bits also gives you -32768.
- **C++**: On parallel architectures, which is most of them now that most CPUs have multiple cores, the C++ "memory model"—that is, the guarantees C++ offers about when code running on one core can actually see the effects of code on other cores—is... interesting.

If my memory serves (and if this hasn't changed in the last, oh, 20 years), Java has an unusual and refreshing shortage of undefined behaviour, which was probably motivated by the goal of "mobile code": a Java program should run anywhere with the same behaviour.

Exercise 7. Do some digging (a few Google searches may be enough) and read about undefined behaviour in your favourite (or least favourite) language. If you find something interesting, surprising, or horrifying, and you probably will, post a note on Piazza.

Exercise 8. (Not an exercise you can expect to actually *do*; just something to think about.) Suppose your programming language allows you to spawn threads that communicate with each other. How would you write an evaluation semantics for such a language?

5 Error rules, small-step semantics and evaluation contexts

5.1 Topics discussed

- a *judgment* is whatever is in the conclusion of a rule, not necessarily $e \Downarrow v$, not necessarily about evaluation or dynamic semantics
- notation for judgments and rules
- error rules for Fun... and more... and still more
- *small-step semantics*
  - inside an evaluation derivation, little steps of computation happen
  - we can model these with a different judgment
  - *reduction rules* for the actual steps of computation
  - rule Step-context and *evaluation contexts* for finding the subexpression where the actual step can happen
- BNFs not just for concrete syntax; can also define subsets of abstract syntax
- evaluation contexts: define with a BNF, notation for replacing holes
- **note:** "evaluation semantics" $\Longrightarrow$ "big-step semantics"
- only need one small-step rule for free variable errors

  (**the lecture on Friday, 2015/10/02 ended around this point**)
- relating small- and big-step semantics
- small-step semantics and recursion

$\boxed{e \Downarrow v}$ Expression e evaluates to value v

$$\frac{}{(\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{add } e1\ e2) \Downarrow (\text{Num } (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{sub } e1\ e2) \Downarrow (\text{Num } (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Eval-with}$$

$$\frac{}{(\text{lam } x\ e1) \Downarrow (\text{lam } x\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\text{lam } x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\text{app } e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

$$\frac{[(\text{rec } u\ e)/u]e \Downarrow v}{(\text{rec } u\ e) \Downarrow v}\ \text{Eval-rec}$$

$\boxed{e\ \text{free-variable-error}}$ e raises a free variable error

$$\frac{}{(\text{id } x)\ \text{free-variable-error}}\ \text{FVerr-id}$$

Figure 5.1 Evaluation rules for Fun

5.2 Collected rules for Fun (again)

Figure 5.1 collects all the rules, again, showing Eval-app-value rather than Eval-app-expr. I added Eval-rec because it will come in handy.

In this note I'll keep using the old Fun language rather than the Fun++ language on the assignment, for tedious reasons (I might give away part of the solution) but also to keep the number of rules small.

Some additional features of this figure are explained below.

5.3 Judgments

Now that you have some familiarity with evaluation rules, you may have room in your mind for more terminology.

We started out (in the AE and WAE languages) by saying that "$e \Downarrow n$" meant "e evaluates to the number n". After we moved on to the Fun language, we needed expressions to evaluate to either numbers or lams, so we stopped using $e \Downarrow n$ and started using $e \Downarrow v$ instead.

Rules and derivation trees, however, are not limited to statements about what an expression evaluates to. Gentzen invented the tree notation to represent mathematical proofs, and its invention substantially predates the development of programming language semantics. The statements "$e \Downarrow n$" and "$e \Downarrow v$" are just particular kinds of *judgments*, and "$e \Downarrow n$" and "$e \Downarrow v$" are particular *judgment forms*.

We actually introduced another judgment form without much fuss: "e free-variable-error", used as the conclusion of the rule Eval-free-identifier (which I'm going to rename FVerr-id). In Figure 5.1, we move this rule away from the others, and add headings like this:

judgment *reading*

Below this heading, we put all the rules whose conclusion has the stated judgment form.

The *reading* is both a "pronunciation key" (how to *read aloud* a judgment of this form), and a suggestion for how to *understand* judgments of this form. The judgment form e free-variable-error is rather self-explanatory, but here are some other judgment forms seen in the wild:

$$\begin{array}{l} \mathsf{P}\ \mathsf{true} \\ \Gamma \vdash \mathsf{P}\ \mathsf{valid} \\ \tau : \kappa \\ \Gamma \vdash e : \tau \\ \Gamma \vdash e :_{\varphi} A \hookrightarrow M \\ e \longrightarrow e' \\ G_1 \vdash^{p}_{\omega} e \Downarrow G_2; t \end{array}$$

We'll learn what some of these mean later in 311. Defining a language via rules usually requires several different judgment forms, both for the dynamic semantics and for the static semantics.

All of the terminology (rule, premise, conclusion, derivation tree, derivable), as well as techniques for working with rules and derivations, such as the "method of hope", is the same for any judgment form, no matter what the purpose of the judgment is. We'll see this in a little while when we introduce *small-step semantics* as a different way of specifying the behaviour of an interpreter, and later when we look at static semantics, particularly static typing.

5.4 Error rules for Fun

Ignoring errors, such as free identifiers, trying to add a function to a number, or trying to apply a number to a function, our rules match the interpreter we wrote. However, the specification of errors isn't very satisfactory.

Let's consider *only* the free identifier (or free variable) error, and whether our interpreter is sound and complete given that rule.

Last time, we said:

Definition 9. Completeness of the interpreter: If $e \Downarrow v$ is derivable then `(interp e)` $= v$.

Definition 10. Soundness of the interpreter: If `(interp e)` $= v$ then $e \Downarrow v$ is derivable.

These notions make sense for other judgments, such as the e free-variable-error judgment:

Definition 11. Completeness of the interpreter's handling of free variable errors:
If e free-variable-error is derivable then `(interp e)` raises a free variable error.

Completeness means that, whenever the rules for deriving e free-variable-error say that a free variable error has occurred, our interpreter will flag such an error. By "raises" or "flags", I mean what the Racket (actually PLAI) `error` function does.

Definition 12. Soundness of the interpreter's handling of free variable errors:
If `(interp e)` raises a free variable error then e free-variable-error.

Is our interpreter's handling of free variable errors sound and complete?

If we evaluate `(interp (id 'i))` our interpreter evaluates `(error "free-variable")`. This seems to correspond to the one rule that can derive the e free-variable-error judgment form. Note that `interp`'s **type-case** branch

```
[id (x)
     (error "free-variable")]
```

doesn't inspect `x`, so it will do the same thing for `(interp (id 'zzzzz))` or any other symbol.

However, our interpreter is *not* sound, because our interpreter is doing a *much better* job of catching free variables! For example:

```
(interp (add (num 5) (id 'i)))
```

gives a free variable error, which is what we *want*, but isn't what the rules say! The rules say that the only expression that should cause a free-variable-error is id. Not an expression that contains an id, but exactly an id.

When we say our interpreter is not sound, we must understand that this does not necessarily mean our interpreter is wrong! Soundness is always *with respect to* a definition. Here, it is our *definition* that really needs to change, because our definition doesn't match our expectation. We expect that a free variable should cause an error even if it's hiding inside an add.

How should we fix our definition? We have to add lots of new rules, like this:

$\boxed{e \text{ free-variable-error}}$ e raises a free variable error

$$\frac{}{(\text{id } x) \text{ free-variable-error}} \text{ FVerr-id}$$

$$\frac{e1 \text{ free-variable-error}}{(\text{add } e1\ e2) \text{ free-variable-error}} \text{ FVerr-add-left} \qquad \frac{e2 \text{ free-variable-error}}{(\text{add } e1\ e2) \text{ free-variable-error}} \text{ FVerr-add-right}$$

$$\frac{e1 \text{ free-variable-error}}{(\text{app } e1\ e2) \text{ free-variable-error}} \text{ FVerr-app-left} \qquad \frac{e2 \text{ free-variable-error}}{(\text{app } e1\ e2) \text{ free-variable-error}} \text{ FVerr-app-right}$$

We would need several more rules, which I won't bore you with, but we have to know when to stop. The following rule would not be helpful, even though it seems to follow the pattern established above.

$$\frac{e \text{ free-variable-error}}{(\text{lam } x\ e) \text{ free-variable-error}} \text{ FVerr-lam??}$$

Why? Well, we want to flag *free* identifiers, but x will be flagged as free even though the lam binds it!

$$\frac{\dfrac{}{(\text{id } x) \text{ free-variable-error}} \text{ FVerr-id}}{(\text{lam } x\ (\text{id } x)) \text{ free-variable-error}} \text{ FVerr-lam??}$$

Putting in this rule seems bad. Fortunately, we can leave it out without affecting soundness and completeness. Our interpreter doesn't catch free variable errors until it actually reaches them:

```
> (interp (lam 'a (id 'b)))
(lam 'a (id 'b))
> (interp (app (lam 'a (id 'b)) (num 0)))
⊗ free-variable
```

Whether this is what we really want is another question; for the sake of stability, to keep us from changing both the interpreter and the rules at the same time, let's just assume it is.

But something still doesn't match up:

$$\frac{e2 \text{ free-variable-error}}{(\text{add } e1\ e2) \text{ free-variable-error}} \text{ FVerr-add-right}$$

This rule is looking too hard: if $e1$ doesn't evaluate to something—for example, when $e1$ is $(\text{rec } u\ (\text{id } u))$—FVerr-add-right will raise an error, even though our interpreter wouldn't.

$$\frac{e1 \Downarrow v1 \qquad e2 \text{ free-variable-error}}{(\text{add } e1\ e2) \text{ free-variable-error}} \text{ FVerr-add-right-better}$$

This should match our interpreter. But we also have to change FVerr-app-right, and other FVerr rules we didn't bother to write down. Now imagine doing this for other errors that our interpreter catches, but that we haven't written rules for. And *then* imagine doing this for an actual language with many more features than Fun!

Our interpreter tries to evaluate the given expression e; this involves interpreting the expressions inside e. But if it notices a free variable, it raises an error. Can we distill what it means to be "about to notice a free variable", and somehow use that in our rules? Yes, but to do this effectively we need to use a different kind of semantics, small-step semantics, which will turn out to have several advantages over evaluation semantics.

(Aside: My opposition to "interpreter semantics" doesn't mean I'm against allowing a particular interpreter that seems to do something sensible, like our `interp` function in `dynsem-fun.rkt`, to guide our development of the rules. In the end, a definition should be independent of all of its implementations, but to develop that definition, we should welcome insights from anywhere!)

5.5 Error rules for Fun, in painful detail

When is our interpreter about to notice a free variable? That is, when we do `(interp e)`, when will our interpreter raise an error?

(1) when e is $(\text{id } \cdots)$

(2) when e is $(\text{add } e1\ e2)$ and we're about to notice a free variable in $e1$

(3) when e is $(\text{add } e1\ e2)$, we evaluated $e1$ to some value, and we're about to notice a free variable in $e2$

(4) same as (2) and (3), but for sub

(5) when e is $(\text{app } e1\ e2)$ and we're about to notice a free variable in $e1$

(6) when e is $(\mathsf{app}\ e1\ e2)$ and $e1$ evaluates to $(\mathsf{lam}\ x\ eB)$ and we're about to notice a free variable in $e2$

(7) when e is $(\mathsf{app}\ e1\ e2)$ and $e1$ evaluates to $(\mathsf{lam}\ x\ eB)$ and $e2$ evaluates to $v2$ and we're about to notice a free variable in $[v2/x]eB$

(8) when e is $(\mathsf{with}\ x\ e1\ eB)$ and we're about to notice a free variable in $e1$

(9) when e is $(\mathsf{with}\ x\ e1\ eB)$, we evaluated $e1$ to some value, and we're about to notice a free variable in eB

This attempted definition is essentially repeating all the evaluation rules, but with "about to notice a free variable" replacing one premise. We have the evaluation rule

$$\frac{e1 \Downarrow (\mathsf{lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\mathsf{app}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

and we could write these rules corresponding to (5)–(7):

$$\frac{e1\ \mathsf{free\text{-}variable\text{-}error}}{(\mathsf{app}\ e1\ e2)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{FVerr-app-1} \qquad \frac{e1 \Downarrow (\mathsf{lam}\ x\ eB) \qquad e2\ \mathsf{free\text{-}variable\text{-}error}}{(\mathsf{app}\ e1\ e2)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{FVerr-app-2}$$

$$\frac{e1 \Downarrow (\mathsf{lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB\ \mathsf{free\text{-}variable\text{-}error}}{(\mathsf{app}\ e1\ e2)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{FVerr-app-3}$$

This is "straightforward" but tedious. Eval-app-value covers the situation where "everything works", that is, when evaluating $e1$ gives a lam, when evaluating $e2$ gives a value, and when evaluating the body under substitution gives a value. The FVerr-app- rules have to deal with every possible point of failure: $e1$ can't be evaluated (FVerr-app-1), $e1$ is fine but $e2$ can't be evaluated (FVerr-app-2), or $e1$ and $e2$ are fine but $[v2/x]eB$ can't be evaluated.

Exercise 13. Write an FVerr rule for rec, and rules for with.

At this point (especially if you did the exercise), you might be thinking, "Curse Gentzen and his 'rules'! I wish I could just raise an error and have it somehow propagate through the rules, like I can use `error` in `interp`!" Sadly, that isn't feasible. While 311 is skipping the mathematical foundations underlying rules and derivations, those foundations exist and are essential to being able to prove properties of rules, and I don't see how the foundations would survive trying to add exceptions *to the rules mechanism itself*.

Switching to a different semantics will help, though.

5.6 Small-step semantics

Forget about error handling for a moment:

When we use the method of hope to derive an evaluation judgment $e \Downarrow v$, we see expressions becoming "more evaluated". Expressions are replaced with values; bound variables in lam and with get replaced with values. (If we used the expression strategy instead of the value strategy, the bound variables are replaced with expressions that aren't necessarily values; still, at least they're known

expressions.)

$$\dfrac{\ldots \qquad \dfrac{(\mathsf{add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2)) \Downarrow \_\_\_ \qquad (\mathsf{Num}\ 10) \Downarrow \_\_\_}{\left(\mathsf{sub}\ (\mathsf{add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 10)\right) \Downarrow \_\_\_}\ \text{Eval-sub}}{\left(\mathsf{app}\ \left(\mathsf{lam}\ x\ \left(\mathsf{sub}\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 10)\right)\ (\mathsf{Num}\ 1)\right)\right) \Downarrow \_\_\_}\ \text{Eval-app}$$

Little steps of computation happen along the way, such as when we replace n_1 and n_2 in "$n_1 + n_2$" and add the actual numbers together.

$$\dfrac{\ldots \qquad \dfrac{(\mathsf{add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2)) \Downarrow (\mathsf{Num}\ 3) \qquad (\mathsf{Num}\ 10) \Downarrow \_\_\_}{\left(\mathsf{sub}\ (\mathsf{add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 10)\right) \Downarrow \_\_\_}\ \text{Eval-sub}}{\left(\mathsf{app}\ \left(\mathsf{lam}\ x\ \left(\mathsf{sub}\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 10)\right)\ (\mathsf{Num}\ 1)\right)\right) \Downarrow \_\_\_}\ \text{Eval-app}$$

This suggests that instead of saying the meaning of an expression is what it evaluates to, we can think of the meaning of an expression as *the expression we get after doing "one step" of computation*. In this way, the meaning of $\left(\mathsf{sub}\ (\mathsf{add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 10)\right)$ is

$$\left(\mathsf{sub}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 10)\right)$$

and the meaning of *that* expression is

$$(\mathsf{Num}\ -7)$$

This might seem tedious (and risking Zeno's paradox?) but we'll be able to recover the same notion of meaning we had with $e \Downarrow v$ (and we won't need a cavalcade of error rules).

The new judgment form will be

$$e1 \longrightarrow e2$$

and is read "$e1$ steps to $e2$".

The rules for this judgment form will feel different from the Eval rules we used before. Rather than writing rules that recursively evaluate subexpressions, as in Eval-add which evaluates the subexpressions $e1$ and $e2$ before doing its actual work (adding n_1 and n_2), most of our rules will only work directly when their subexpressions are already values. Just one rule (backed by a BNF definition) will manage "finding" a subexpression to step.

First, let's write the "basic" rules, sometimes called *reduction rules*. Think of these rules as where computation really happens. We show these in Figure 5.2.

The last rule in Figure 5.2 is not a reduction rule, and is quite concise but depends on another definition, which we'll explain next: *evaluation contexts*.

The idea is that if we are given an expression that contains a subexpression e, where e is "an expression we should step next", and we can use one of the basic rules (reduction rules) to step e to e', then the whole expression steps to $\mathcal{C}[e']$. The expression we get after stepping is the same as the one we started with, except for the part e that just "took a step".

To define what $\mathcal{C}$ means, we'll use a BNF. We've been using BNFs to define the concrete syntax of languages, but BNFs are versatile and can also be used with *abstract* syntax. To (hopefully) clarify that this BNF is describing abstract syntax, not concrete syntax, I'll follow the convention I've been using in the rules, where we write e, v, n, etc. rather than using angle brackets $\langle \mathrm{E} \rangle$ (as PLAI and the Algol 60 Report do for concrete syntax).

This is also an opportunity to define values v using a BNF:

$$\begin{array}{lrl} \text{Values} & v & ::= (\mathsf{Num}\ n) \\ & & \mid (\mathsf{lam}\ x\ e) \end{array}$$

$\boxed{e1 \longrightarrow e2}$ Expression $e1$ steps to $e2$

Reduction rules:

$$\frac{}{(\mathsf{add}\ (\mathsf{Num}\ n_1)\ (\mathsf{Num}\ n_2)) \longrightarrow (\mathsf{Num}\ n_1{+}n_2)}\ \text{Step-add}$$

$$\frac{}{(\mathsf{sub}\ (\mathsf{Num}\ n_1)\ (\mathsf{Num}\ n_2)) \longrightarrow (\mathsf{Num}\ n_1{-}n_2)}\ \text{Step-sub} \qquad \frac{}{(\mathsf{app}\ (\mathsf{lam}\ x\ eB)\ v) \longrightarrow [v/x]eB}\ \text{Step-app-value}$$

$$\frac{}{(\mathsf{with}\ x\ v1\ e2) \longrightarrow [v1/x]e2}\ \text{Step-with} \qquad \frac{}{(\mathsf{rec}\ u\ e) \longrightarrow [(\mathsf{rec}\ u\ e)/u]e}\ \text{Step-rec}$$

Context rule:

$$\frac{e \longrightarrow e'}{\mathcal{C}[e] \longrightarrow \mathcal{C}[e']}\ \text{Step-context}$$

Figure 5.2 Small-step semantics

Exercise 14. Update this definition of values v to reflect Fun++ (the Assignment 2 language).

Now, the definition of evaluation contexts:

$$\begin{aligned}\text{Evaluation contexts}\quad \mathcal{C} ::= {} & [\,] \\ & \mid (\mathsf{add}\ \mathcal{C}\ e) \\ & \mid (\mathsf{add}\ v\ \mathcal{C}) \\ & \mid (\mathsf{sub}\ \mathcal{C}\ e) \\ & \mid (\mathsf{sub}\ v\ \mathcal{C}) \\ & \mid (\mathsf{app}\ \mathcal{C}\ e) \\ & \mid (\mathsf{app}\ v\ \mathcal{C}) \\ & \mid (\mathsf{with}\ x\ \mathcal{C}\ e)\end{aligned}$$

The empty brackets $[\,]$ are called a "hole". Some examples of evaluation contexts:

$$\begin{aligned}&(\mathsf{add}\ [\,]\ (\mathsf{app}\ e3\ e4)) \\ &(\mathsf{sub}\ (\mathsf{Num}\ 5)\ [\,]) \\ &(\mathsf{app}\ (\mathsf{app}\ [\,]\ e1)\ e2))\end{aligned}$$

In all of these, the hole $[\,]$ appears in a position where we should try to step:

- to add, we need two values (numbers), so we should try to step the first subexpression
- to sub, if we have a value $(\mathsf{Num}\ 5)$ as the first subexpression we should try to step the second subexpression
- to apply a function to an argument, where the function is itself a function application, we need to step inside that application

We also need some more notation: we want to use these contexts in our rule Step-context, but a context $\mathcal{C}$ isn't an expression because it always has a hole [] in it, and holes aren't in our abstract syntax! (They're also not in our concrete syntax, which is just as well.) So we'll write

$$\mathcal{C}[e]$$

to mean the context $\mathcal{C}$ with its hole replaced by e. This is best understood through examples, so:

if $\mathcal{C}$ is...	and e is...	then $\mathcal{C}[e]$ is...	
(add [] (app e3 e4))	(sub e1 e2)	(add (sub e1 e2) (app e3 e4))	
(add [] (app e3 e4))	(Num 1)	(add (Num 1) (app e3 e4))	*
(sub (Num 5) [])	(app (lam x (id x)) (Num 2))	(sub (Num 5) (app (lam x (id x)) (Num 2)))	
(app (app [] e1) e2)	(add (lam ...) (lam ...))	(app (app (add (lam ...) (lam ...)) e1) e2)	**

The second example (marked $*$) illustrates that contexts $\mathcal{C}$ don't care whether the expression replacing the hole can actually step: (Num 1) is a value, so it won't step to anything. We wouldn't be able to apply Step-context on this particular $\mathcal{C}[e]$, but the definition of $\mathcal{C}$ doesn't care. The last example (marked $**$) also shows this: (add (lam ...) (lam ...)) won't step, but it can still replace the hole. The definition of $\mathcal{C}$ is all about finding a *position* within the expression; $\mathcal{C}$ doesn't care what's at that position.

5.6.1 Error handling

Now for the thrilling conclusion: With a small-step semantics, the annoying error rules can be replaced with just one:

$$\frac{}{\mathcal{C}[(\text{id x})]\ \text{free-variable-error}}\ \text{FVerr-context}$$

5.6.2 Relating small- and big-step semantics

You've seen small-step semantics now, so I can tell you that "evaluation semantics" is often called "big-step semantics": to evaluate (add e1 e2), evaluation semantics has e1 take a "big step" and then has e2 take a "big step".

Have we accidentally defined a different language? Hopefully not. First we need a useful definition:

Definition 15. We write $e \longrightarrow^* e'$ to mean that e takes 0 or more steps to e'. (That is, either $e = e'$, or $e \longrightarrow e2$ and $e2 \longrightarrow^* e'$.)

If we're really enjoying Gentzen's notation, we can write this definition using rules:

$\boxed{e \longrightarrow^* e'}$ Expression e takes 0 or more steps to e'

$$\frac{}{e \longrightarrow^* e}\ \text{Steps-zero} \qquad \frac{e \longrightarrow e2 \qquad e2 \longrightarrow^* e'}{e \longrightarrow^* e'}\ \text{Steps-step}$$

Exercise 16. Write rules deriving $e \longrightarrow^+ e'$ such that $e \longrightarrow^+ e'$ if and only if e takes *one* or more steps to e'. You can use other judgment forms, including $\longrightarrow^*$, as premises.

Now the following should hold:

(1) If $e \Downarrow v$ then $e \longrightarrow^* v$.

(2) If $e \longrightarrow^* v$ then $e \Downarrow v$.

Proving these is outside the scope of 311, but they should hold unless I've made a mistake...

A consequence of (1) and (2) is that, ignoring the whole issue of error handling, we can write an interpreter either by following the Eval rules or by following the Step rules. In the former case, we can look at the interpreter code and (hopefully) see that it directly implements the Eval rules, and then appeal to (1) and (2) to show that our interpreter (indirectly) follows the Step rules. In the latter case, we have to write an interpreter (maybe we should call it a stepper?) that directly implements the Step rules, and again appeal to (1) and (2), to show that we have (indirectly) implemented the Eval rules.

Both small- and big-step semantics are used to define programming languages, but small-step has some important advantages, (partly) explained below. (Big-step is usually considered easier to understand, which is one of the reasons I presented it first.)

5.6.3 Small-step semantics and recursion

As with big-step semantics ($e \Downarrow v$), attempting to step $(\mathsf{rec}\ \mathfrak{u}\ (\mathsf{id}\ \mathfrak{u}))$ won't get us anywhere useful. However, small-step "fails" less catastrophically. In big-step, we attempted to construct an infinite derivation tree; in small-step, we can apply rule Step-rec to derive a perfectly good judgment:

$$\frac{}{(\mathsf{rec}\ \mathfrak{u}\ (\mathsf{id}\ \mathfrak{u})) \longrightarrow (\mathsf{rec}\ \mathfrak{u}\ (\mathsf{id}\ \mathfrak{u}))}\ \text{Step-rec}$$

If we keep stepping, we won't get anywhere. If we enjoy Gentzen's notation and regard the Steps-zero and Steps-step rules as the definition of "stepping 0 or more times", we will again attempt to construct an infinite derivation tree.

Being able to talk about taking *one* step is useful. For example, an *information-flow type system* can prove that a given program will never leak secret information (e.g. by printing a cleartext password). For a big-step semantics, this property cannot be stated easily: we can certainly model what evaluation prints, through a judgment

$$e \Downarrow v\ \mathsf{prints}\ s$$

read "e evaluates to value v and prints the string s". But we can't prove the following statement:

> For all e such that e passes the information-flow type system,
> either $e \Downarrow v$ prints s where s contains no secret information,
> or e free-variable-error.

We can't prove this because evaluating e might *diverge*, for example, if e is $(\mathsf{rec}\ \mathfrak{u}\ (\mathsf{id}\ \mathfrak{u}))$. We would instead have to prove the following (using "$e \Uparrow$" to mean e diverges):

> For all e such that e passes the information-flow type system,
> either $e \Downarrow v$ prints s where s contains no secret information,
> or e free-variable-error,
> or $e \Uparrow$ but e has not yet printed any secret information.

But what does "e has not yet printed any secret information" mean? The judgment $e \Downarrow v$ prints s is supposed to define what expressions should print which strings, but it only makes sense if e has finished and returned a value.

In small-step semantics, we still have a notion of diverging expressions: e diverges if there exists no value v such that $e \longrightarrow^* v$. But we can easily talk about the steps we take as we go:

For all e such that e passes the information-flow type system,
either e is a value,
or e free-variable-error,
or $e \longrightarrow e'$ prints s where s contains no secret information.

We could then show (by induction on the number of steps) that for any number of steps, no secret information will be printed.

Another argument for small-step semantics is that, while we would like Fun(++) programs to terminate and return a value, not all programs should terminate. An operating system kernel[1] or web server should, in principle, run forever. We still want to know that the web server will never send sensitive information in the clear (cf. Heartbleed), which is analogous to our information-flow example above.

[1] I have an ancient Toshiba laptop that stayed up for *over three years*, running OpenBSD. I had to turn it off when I moved back to Canada; the flights here were a little longer than its battery life.

6 Taxonomy of languages

6.1 Topics discussed

- categorizing syntax: "formal languages"; "C-like"; Algol-60
- categorizing semantics
- orthogonal language features

6.2 Categorizing languages: syntax

A language consists of syntax and semantics. Semantics consists of dynamic semantics (perhaps defined using a big-step semantics $e \Downarrow v$ or a small-step semantics $e1 \longrightarrow e2$) and static semantics. We'll jump into static semantics later this week.

I'm not very interested in syntax. However, syntax can be classified (somewhat) usefully using the theory of *formal languages*, in which "language" refers only to syntax: a "language" in that sense is simply a set of strings that are considered syntactically valid. Languages, or rather syntaxes, can be organized into the Chomsky hierarchy, first with *regular languages*, then *context-free languages*, then *context-sensitive languages*, and finally unrestricted languages. Each level in the hierarchy has a corresponding kind of automaton that *recognizes* that language, that is, the automaton determines whether the string is in the language (is syntactically valid). (Finite automata can recognize regular languages, pushdown automata can recognize context-free languages...) This is of relatively little interest to me, partly because even large programming languages have context-free syntax.

"BNF" is a specific notation for writing a context-free grammar.

(This general rule that PLs have context-free syntax has some exceptions. Parsing Perl is not context-free, because it's undecidable: `http://www.perlmonks.org/?node_id=663393`.)

Less formally, categories such as "C-like syntax" and "Lisp-like syntax" are easily recognized by humans, but carry little information: C, Java, and JavaScript all have C-like syntax, but their *semantics* are extremely different.

Algol-60 had the idea, now largely forgotten, of explicitly distinguishing the notation used in the language definition from the notation that programmers would type in (to the punch-card writer). This was partly because there was no broadly recognized standard character encoding—ASCII wasn't defined until 1963—but it meant that Algol-60 didn't try to forbid programmers from using certain "reserved words" or "keywords", because the language's creators assumed that each Algol compiler would define its own mapping from the "local" notation to the Algol notation used in the report.

This idea was spurred by a violent disagreement about decimals; see Wexelblat, *History of Programming Languages (I)* (1981), p. 126. If this idea hadn't been forgotten, we might now be using

editors and IDEs that could automatically switch between keywords in English and keywords in other languages. And perhaps between decimal separators.

6.3 Categorizing languages: semantics

If we can't usefully categorize syntax, what about semantics?

Yes, but maybe not with the usual categorization. Let's try the usual one anyway.

Procedural	Object-oriented	Functional	Logic
Fortran	Simula	Racket	Prolog

3 Categorizing languages: semantics

If we can't usefully categorize syntax, what about semantics?

Yes, but maybe not with the usual categorization. Let's try the usual one anyway.

Procedural	Object-oriented	Functional (First-class functions)	Logic
COBOL		Lisp Scheme	Mercury
Fortran	Simula-67	Racket	Prolog
C	Java	JavaScript	
C++?	C++	Haskell	
Algol-60	Smalltalk(1970s)	SML, OCaml	
Pascal	Ruby	Python	
Modula-2,-3?	C#	Scala	
Python?	JavaScript?	Swift?	
bash	Scala	F#	
awk?	OCaml	C++	
BASIC	Swift	Java	
Visual BASIC?	F#?	Ruby	
(assembly)		Rust?	
Rust?	Rust?		

SQL?

was partly because there was no broadly-recognized standard character encoding—ASCII wasn't defined until 1963—but it meant that Algol-60 didn't try to forbid programmers from using certain "reserved words" or "keywords", because the language's creators assumed that each Algol compiler would define its own mapping from the "local" notation to the Algol notation used in the report. (The actual reason they adopted this idea: a violent disagreement about decimals; see Wexelblat 1981, *History of Programming Languages (I)*, p. 126). If this idea hadn't been forgotten, we might now be using editors and IDEs that could automatically switch between keywords in English and keywords in other languages. And perhaps between decimal separators.)

Categorizing languages: semantics

If categorizing syntax isn't that interesting, can we categorize semantics?

Yes, but maybe not with the usual categorization. Let's try the usual one anyway.

Imperative	Object-oriented	Functional	Logic
Fortran	Simula-67	Racket	Prolog
C ?Java	Java	Haskell Mercury	Mercury
?COBOL	JavaScript	?Perl	Twelf
assembly	C++	?JavaScript	
Algol-60	Python	?Python	
Algol-68	C#	Java 8 C++11	
Pascal	Smalltalk	?C#	
Modula-2	Objective-C	F#	
GLSL	Ruby	Ruby	
?Shell	Swift	Standard ML	
BASIC	Scala	OCaml	
C++	OCaml	Scala	
	Self	Smalltalk	
		~~WAE~~	
		~~AE~~	
		Fun	
		Fun++	

6.4 Categories: fuzzy at best

Essentially no languages fit neatly into these categories.

Explicitly "hybrid" languages, like OCaml (functional + OO), Mercury (functional + logic), should be expected to fall into more than one category, but really, no language fits neatly into these categories.

Everyone agrees that Racket is functional, but it has mutable data—but mostly not by default. Same for Standard ML and OCaml. In Lisp, mutable data is (was?) default, but Lisp otherwise "feels" functional?

Haskell doesn't have mutable data... but a lot of machinery, idioms, and libraries have been developed (and are extensively used) to let Haskell programmers pretend that it does!

OCaml has objects, but you don't have to use them and lots of OCaml programmers never do

C and C++ have everything mutable by default, but you can write `const`.

Java has "base types", like `int`, that aren't object-oriented at all.

Some OO languages (Self, Go?, ...?) don't have classes.

So these categories are really about default behaviours, and (even more) about what programmers actually do most of the time:

- C programmers tend to use mutation, even when they don't have to.
- Racket, SML, OCaml programmers tend to avoid mutable state "by default".
- OCaml programmers tend to avoid using objects.
- Java programmers (I assume?) tend to use objects even when they could use base types.
- Curiously, Haskell programmers seem pretty fond of the machinery developed to let them pretend that Haskell has mutable state...

6.5 Categorizing particular language features

Accepting that the categories listed above only suggest a kind of probability distribution on whether a particular feature is present, or how a particular feature works, we can instead ask smaller, more specific questions, like "what evaluation strategy does language X use?"

But we should be aware that we're probably asking "what is the *default* evaluation strategy in language X?" For example, Scala lets you use the expression strategy, but only if you explicitly ask. Racket and SML use the value strategy, but it's not hard to simulate the expression strategy, just slightly annoying. Haskell uses lazy evaluation (related to the expression strategy), but you can get the value strategy by explicitly asking for it.

With that caveat, we can ask specific questions about "real" mainstream (and, usually, imprecisely defined) languages, and learn something (even if it's not truly precise) by comparing their features to the "equivalent" features in a small, idealized language like Fun++.

(I *would* be very comfortable putting Fun and Fun++ under the "Functional" heading.)

6.5.1 Categorizing Fun's variables

In addition to the value strategy (commonly known as "call by value") and expression strategy (commonly known as "call by name") for functions, we can consider whether Fun[++] has mutable state. Here, we can comfortably say it doesn't, because we can (and have) defined the dynamic semantics of Fun using substitution. When we substitute for the identifier bound by a lam, that identifier disappears, leaving no trace of its existence. In the body after substitution, we can't tell which expressions resulted from substituting for x and which expressions didn't. When we apply

```
(Lam x (Add (Id x) (Num 1)))
```

to (Num 1), substitution gives

```
(Add (Num 1) (Num 1))
```

which has no sign of which (Num 1) was originally (Id x).

So we can conclude that Fun's variables (identifiers) are *immutable*, and that adding mutable variables would require us to change the semantics. (We will almost certainly do this later in 311.)

6.5.2 Categorizing Fun's functions

Functions are classified according to how "first-class" they are—essentially, whether they are values that can be passed around and used to construct other values (like integers can be passed around and used to construct other integers, or binary trees can be passed around and used to construct other binary trees).

("First-class" is standard terminology, but misleading, because it suggests that a "first-class function" is somehow special, when it's really the opposite: a first-class function is *a completely ordinary value*. It's languages without first-class functions that have separated their functions from the rabble of ordinary values.)

Here, we again have a clear answer: functions in Fun are values, with no restrictions: they can be passed as arguments to other functions, and returned as results. As we add features to Fun, we should verify that they've kept this status. For example, the pairs in Fun++ don't affect this status, because pairs can hold any values, including functions.

When two language features do not interfere with or affect each other, we can say they are *orthogonal*, by a kind of geometric analogy. This is a vague definition, but I don't know of a better one. More

precisely, we can say that two features are *defined orthogonally* if their definitions are all independent. (Warning: I *think* this is what other people mean by "defined orthogonally", but I'm not completely sure. I'm pretty sure that this is a *useful* definition.)

Independence is not an entirely precise notion. Some cases are pretty clear: in Fun, functions and numbers are defined independently, because none of the rules for numbers and arithmetic mention any of the abstract syntax for functions, and none of the rules for `Lam` and `App` mention any of the abstract syntax for numbers.

Some cases are less clear. I *want* to say that in Fun, functions and `Let` are defined independently, because—again—the rules for functions don't mention `Let`, and the rule(s) for `Let` don't mention `Lam` and `App`. You could argue that both functions and `Let` depend on the definition of substitution... which mentions *all* the variants of the abstract syntax.

Exercise 17. Which new features in Fun++ (Assignment 2) are independent of which other features? (Because independence is not really precise, no answer can be perfectly right or perfectly wrong.)

Earlier, I almost said "when two language features do not interact with each other", but that could be misleading. Fun++ allows you to mix functions and pairs as much as you like: a function can take a pair as an argument, the pair can contain functions, and you can return a pair of functions. This demonstrates a key benefit of orthogonal designs: the combination of features, though defined separately, leads to a language that "subsumes" (maybe with a little added sugar) other, non-orthogonal features. If you can pass pairs as arguments, you are very close to allowing functions that take multiple arguments. But you got there by starting with the simplest possible form of function (a single argument to a single result), rather than saying, "well, we'll make functions take different numbers of arguments, so we have to think about whether a given function is being called with the right number of arguments...".

Exercise 18. Given Fun plus pairs, how would you add multiple-argument functions as syntactic sugar?

Exercise 19. Given Fun without pairs, could you add multiple-argument functions as syntactic sugar? If so, how?

7 Static semantics: Types

"You must always ask yourself: What kind of an animal is it? Is it a function? Is it a set?"
—Prof. Maria Balogh

7.1 Topics discussed

- why use types?
  - stop bad things from happening
  - make sure that good things will happen
- classifying errors
- catching errors statically vs. dynamically
- typed vs. untyped languages; safe vs. unsafe languages:
- catching bugs with Haskell's type checker; not catching bugs with Haskell's type checker
- refined type systems
- object-oriented languages
- disadvantages of typed languages

7.2 "Static" vs. "dynamic"

A language consists of syntax and semantics. Semantics consists of dynamic semantics (perhaps defined using a big-step semantics $e \Downarrow v$ or a small-step semantics $e1 \longrightarrow e2$) and static semantics.

What makes something static or dynamic?

A simplistic model—which mostly made sense in the 1960s—is that "static" means "at compile time", and "dynamic" means "at run time". If our language is implemented by an interpreter, rather than a compiler, this model becomes:

- "static" means "before running the program", and
- "dynamic" means "while running the program".

So in our Fun implementations, "static" would mean "before starting `interp`", and "dynamic" would mean "while running `interp`".

Under this model, we can say that, in Fun:

- syntax checking is static (because it happens inside `parse`, which runs before `interp`); and
- arithmetic is dynamic because numbers are added and subtracted inside `interp`, but not inside `parse`).

Besides arithmetic, many other operations are dynamic, such as function calls and Pair-case.

Most language implementations, especially highly engineered compilers, muddy this model. For example, any "serious" C compiler, given code that looks like

```
i += (15 * 1024);
```

will generate the same machine code as it would for

```
i += 15360;
```

The compiler knows that the result of `15 * 1024` cannot change, so it does the multiplication *at compile time*.

Similarly, I would expect that good C compiler would[1] generate the same code from

```
i += 1024;
i += 1024;
```

as from

```
i += 2048;
```

The compiler knows that adding 1024 twice is the same as adding 2048 once.

```
i += 1024;
if (i == 0) {
  i += 1024;
} else {
  i += 1024;
}
```

I would also expect a good compiler to optimize the above code: the compiler would analyze the "then" and "else" branches, see that they are doing the same thing, and generate code that doesn't test `i` at all. This is a form of *static analysis*.

7.3 Prevention

The most important kind of static semantics is *typing*, sometimes known as *static typing*. We'll see later that typing can be defined through rules.

What's the point of typing? I know two good answers to this question. I like one of these answers better, but first I'll give the more popular answer.

Safety: Typing stops bad things from happening, by telling you that they could happen, and not letting you run your program.

[1] Subject to certain conditions; if I remember correctly, if `i` is declared to be `volatile`, the additions must be done separately.

7.3.1 Errors: a renewable resource

What kinds of bad things can happen in programs? As you know, there are many such animals.

- **Syntax errors**: missing ")", missing semicolon, missing keywords, extra keywords, "illegal string literals", ...
- **Scope errors**: "unbound identifier", "unknown variable", "duplicate definition for identifier", ...

Anything that doesn't match the language's BNF is a syntax error. A program with a scope error matches the BNF, so it's (usually?) not considered a syntax error.

Syntax and scope errors are pretty universal in programming languages. Other kinds of errors depend on the language; a language without arrays, for example, won't have array bounds errors.

- **Agreement errors and "mismatches"**: The terminology, and the specific error messages, for these errors depend very much on the language; here are some examples:

```
- 3 + "a";
stdIn:1.1-1.8 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * string
  in expression:
    3 + "a"
```

```
> (+ 3 "a")
+: contract violation
  expected: number?
  given: "a"
  argument position: 2nd
  other arguments...:
   3
```

```
> ((lambda (x) x) 'a 'b)
⊗ #<procedure>: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 2
  arguments...:
   'a
   'b
```

These are sometimes called *type errors*, but I would like to use that term in a more specific way, so I'll try to avoid using it for now.

- **Array bounds error**: trying to access an element outside an array.
- **Not returning anything**: writing a procedure that's meant to return a value, but doesn't (happens to me in Python).

Each language gets to decide what counts as an error. For example, writing `"a" + "b"` is not an error in Python, but `(+ "a" "b")` is an error in Racket. Writing `(+ 1 0.5)` is not an error in Racket, but `1 + 0.5` is an error in SML (it doesn't let you mix integers and floats).

Division by zero is almost always an error—but if you really wanted to, and if you have no respect for algebra, you could define a language in which division by zero returned zero.

Integer overflow is an error in many languages, but not all. Some earlier lecture notes discussed C's overflow behaviour; to summarize, overflow on C's `unsigned int` is supposed to "wrap around", but C doesn't specify what should happen if you overflow on a (signed) `int`. We could reasonably say that a C program that overflows a signed `int` has a bug, since the definition of C doesn't specify what that program will do, but C does not specify that it is an error.

An error is a failure that is somehow *caught* and reported, though not necessarily reported in a clear or helpful way. Thus, in Python, not `returning` from a function is not an error: the function will return `None`. This is well-defined, but not what I wanted to do. In C, not `returning` returns an unspecified value (I think?); again, not an "error".

7.3.2 Warnings

Language *implementations* often try to help programmers by giving "warnings" for code that is likely to be wrong, but doesn't do anything the language actually forbids. For example, `gcc` has many kinds of warnings, some of which can be extremely useful. To me, many of these warnings are only to compensate for C's design flaws, but some warnings are useful even in languages I like better. OCaml can warn you when you use OCaml's `let` (like Racket's `let`) to bind an identifier that you never use, which catches quite a few bugs.

7.3.3 When are errors caught?

With some idea of *what* an error is—something that is caught, or reported—we can ask: *when* are errors reported?

As usual, it depends on the language, but we can make some generalizations that are (almost) always true:

- **Syntax errors** are caught during parsing.
- **Scope errors** are often caught during parsing, but not always; Racket doesn't catch them until you either run your program or click "Check Syntax" (though DrRacket's "Check Syntax" checks for a few kinds of errors that aren't usually considered syntax errors).

Beyond syntax and scope errors, it depends entirely on the language.

- **Agreement errors and mismatches**: Caught at run time in Racket and Python; caught at compile time in C, SML, OCaml, Haskell. (Java catches many of these at compile time, but not all.)
- **Array bounds error**: Caught at run time in Racket, Python, Pascal, Java, SML, OCaml, Haskell; (mostly) caught at compile time in some "cutting-edge" (last 20 years) research languages.

  C's behaviour is almost entirely undefined; a program reading or writing outside an array may crash ("segmentation fault", "bus error", etc.) or continue unpredictably (or *too* predictably, as with countless viruses that exploit "buffer overruns").
- **Not returning anything**: Caught at run time (sometimes?) in Racket; caught at compile time in Java, SML, OCaml, Haskell. (Not an error in Python, C, C++.)

7.3.4 Types: raising errors earlier than run time

The most popular purpose of a *type system* is to prevent "agreement errors and mismatches", such as applying a list to a function (rather than applying a function to a list) or using + to add things that can't be added. We can specify a type system with rules (in fact, this is much closer to Gentzen's motivation—formal proofs—than using his notation to specify dynamic semantics), which guide the language implementor at compile time, rather than run time.

It doesn't make sense to talk about "compile time" unless there's a compiler. So, to cover both compilers and interpreters, we'll say that types catch errors *statically*, and that a type system is part of the *static semantics* of a language.

The part of a language implementation that checks the abstract syntax of a program, to see whether it violates the type system, is called a *type checker*.

Often, the type checker is checking expressions (or statements, etc.) against type declarations written by programmers. But some languages don't require programmers to write types (or to write only a few types). In these languages (e.g. Haskell), the type checker is sometimes called a *type inferencer*, because it *infers* types the programmer didn't write. The line between "checking" and "inference" is fuzzy, so I use "type checker" for all typed languages, even languages that infer types.

In an interpreter for a typed language, types are checked after parsing (so the type checker can work with abstract syntax instead of concrete syntax), but *before* running the program. In a compiler for a typed language, types are checked before the compiler generates machine code.

7.3.5 What about C?

The C language (and C++) don't fit neatly into the "typed"/"untyped" space. If you've written many C programs, you know that C compilers like to complain about type mismatches—so C must be typed, right?

Without getting mired in terminological disputes, that question has two reasonable answers:

- C is not typed, because a program that passes the type checker may still do (clearly) bad things, such as segfault; and this is unlike Java, SML, and Haskell.
- C is typed (C compilers complain about type errors!), but not *safe*, because (as an example) C never checks for array bounds errors. (Sometimes, C is called "weakly typed".)

If we like the second answer better, we can classify languages along two dimensions: typed vs. untyped, and safe vs. unsafe. Then we would say that

- C is typed but unsafe;
- Java is typed and safe;
- SML, OCaml, and Haskell are typed and safe;
- Racket, JavaScript, and Python are untyped but safe;
- assembly language is untyped and unsafe.

These distinctions are explained pretty well by Luca Cardelli, in the first few pages of a paper (`http://www.lucacardelli.name/Papers/TypeSystems.pdf`). I encourage you to read it (the first few pages, not necessarily the whole paper!), though I can't promise that my terminology will always be the same as his.

Most "scripting languages" are untyped and safe.

Question: What if we implement a safe language using an unsafe language? For example, `bash`, which is safe, is written in C, which is unsafe. What if the `bash` interpreter segfaults?

I wouldn't say this makes `bash` unsafe. Rather, the *implementation* of `bash` has a bug. I haven't read the definition of `bash`, but unless it says that some behaviour is unspecified, any interpreter that segfaults is not consistent with the definition.

This goes for compilers as well: a Racket compiler might have a bug that causes it to generate machine code that segfaults, even though Racket is safe, so programs should never segfault.

(A more subtle case is when a language's implementation *is* consistent with the language's definition, but the language's definition is wrong. For example, a compiler that follows an "obvious" definition of type polymorphism may generate unsafe code. I mentioned *determinism* in previous lectures as an example of a good property of a language's definition; an arguably more important property is *type safety*, which we'll cover later.)

7.4 Good things aren't happening, and I don't like that either

I said that safety—stopping errors from happening, or rather, reporting errors statically (before you run the program) rather than dynamically (while the program is being run, either by you, or by an unhappy user), is the most popular reason to use a typed language.

Another reason, which I think is more interesting, is to ensure that things you *want* to have happen actually will happen.

When you write a helper function, you (should) write a comment with a "signature" that describes what kind of animals the function expects as input, and what kind of animal it produces as output.

;; match-length : string string → natural
;; interleave : (listof any?) (listof any?) → (listof any?)
;; contains-sequence : (list-of symbol?) (list-of symbol?) → boolean?
;; truth-or-lie? : Bool-expr → boolean?

Unfortunately, in Racket, these signatures are merely comments. Racket is not typed, so it doesn't check whether any of these signatures match the function you actually wrote.

You can see some inconsistency in our style, in fact: is it "`listof`" or "`list-of`"? Or maybe we should put a question mark after `Bool-expr`. After all, `Bool-expr?` is an actual Racket/PLAI function, as is `boolean?`. But `natural` isn't (even with a question mark). These signatures are not part of the Racket language, so they are useful documentation, but Racket doesn't check whether that documentation is accurate.

In contrast, in a typed language, the signatures you give actually matter to the language. If we write `interleave` in Haskell:

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:first2:(interleave rest1 rest2)
               -- ^ Haskell ":" is like Racket "cons"
main =
   do
      putStrLn (show (interleave [1, 3, 5, 7] [2, 4, 6, 8, 9]))
```

the *type annotation* (or *type declaration*, or *type signature*) on the first line guarantees that the Haskell type checker will make sure—assuming `interleave` is passed two lists of integers—that every clause of the definition will evaluate to a list of integers. It will also check that, when we call `interleave` on the last line, that we are passing lists of integers. All of this happens after parsing, *not* when we run the program.

When the Haskell program *is* run, it doesn't have to check whether the second argument of `cons` (spelled ":" in Haskell) is a list: it *will* be a list, because the type checker accepted the program. Here, we are still in the category of "stop bad things from happening".

Quite a few mistakes will be caught by Haskell's typechecker. For example, if we wrote

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:first2:(interleave rest2)
```

Haskell will complain, because we applied `interleave` to one argument rather than two. (Actually, it will complain because `(interleave rest2)` returns a function of one argument, and a function of one argument is not a list.)

But many other mistakes will not be caught. We might forget to cons `first2`:

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:(interleave rest1 rest2)
```

This will not be caught: the type annotation demands that `interleave` return something of type `[Int]`, a list of integers, and `first1:(interleave rest1 rest2)` has that type.

All mainstream typed languages (including Haskell and SML) are limited in what their type systems can check. The specific limits vary from language to language. Haskell (or rather its most popular compiler, GHC) has developed a rather powerful, but complicated, type system; SML has a simpler type system than Haskell.

What I said above—that Haskell will check "that every clause of the definition will evaluate to a list of integers"—is not entirely accurate. We will make this kind of statement accurate, and more precise, over the next few weeks. This should also illuminate the line between "preventing bad things" and "ensuring good things".

7.4.1 Refined type systems

While popular typed languages are limited in what their type systems can check, many experimental typed languages push these limits—sometimes amazingly far. What if we could write, in the type annotation for `interleave`, that the length of the list it returns is equal to the sum of the lengths of its arguments? This is within the power of modern Haskell, and of several recent experimental languages.

What if we could write that `interleave` should return a list whose elements are a *permutation* of the elements in its arguments? This is (I believe) beyond Haskell, at least for now.

7.5 Object-oriented languages

Like functional languages, some object-oriented languages are typed, and some are not. The one you're probably most familiar with, Java, is typed. Java's type checker catches many mistakes, but it catches fewer mistakes than Haskell or SML programmers might like. We will explore this in future lectures, but a short, vague explanation is that object-oriented languages assume an "open world": given a particular class, say `DoorLock`, we can declare subclasses of `DoorLock` representing new kinds of locks. We don't know in advance how many subclasses of `DoorLock` will be created. Back in the 1990s, Java was motivated by the desire to send Java programs over the Internet, with the expectation that a program written by the original `DoorLock` author might interact with subclasses of `DoorLock` written by other people around the world.

In contrast, the **define-type** of PLAI, the `datatype` declaration of SML, and the `data` declaration of Haskell define a "closed world": a PLAI program cannot add variants to a **define-type**. This is what allows PLAI to statically check whether you have missed a branch in a **type-case**. In Java, we cannot enumerate all possible subclasses, because our compiled Java program might load new ones!

7.6 Typed programs run faster

Another advantage of typing, which slipped my mind until just now, is that an implementation of a typed language can safely omit some of the checks that would otherwise be required. (Cardelli calls this *economy of execution*.) For example, when you evaluate (Add *e*1 *e*2), you have to check that the values that *e*1 and *e*2 evaluate to are Nums. This remains true even if you use `Num-n` to access the `n` field of the `Num` variant: Racket/PLAI will do this check "under the hood".

In an interpreter, the cost of such checks is almost certainly far outweighed by the difference in cost between interpreted code and compiled code. So this point in types' favour is usually raised in the setting of compiled code. This was a powerful argument for typed languages until the 1990s; modern hardware architectures have drastically reduced the cost of such checks.

7.7 Disadvantages of typed languages

Catching errors statically seems better than catching them dynamically. If your program has an error, you probably want to find out sooner rather than later. But what if the "error" wouldn't have actually happened? Consider the program in Section 7.8.2. Java rejects this program because the `else` branch doesn't have a `return`. However, we would expect that this won't matter, because the test in the `if` statement will always be true and the `then` branch, which *does* have a `return`, will be executed.

Whether this expectation is true is another question. Mathematical properties of the reals rarely hold for floating-point numbers. What if `x` is $+\infty$? If you use floating-point arithmetic for anything important, you probably need to become horribly familiar with the IEEE 754 standard. Wikipedia has the following hint of the horrors lurking within:

> `https://en.wikipedia.org/wiki/IEEE_floating_point`
>
> - Two infinities: $+\infty$ and $-\infty$.
> - Two kinds of NaN: a quiet NaN (qNaN) and a signaling NaN (sNaN). A NaN may carry a *payload* that is intended for diagnostic information indicating the source of the NaN. The sign of a NaN has no meaning, but it may be predictable in some circumstances.

I believe that equality is one of IEEE 754's unpredictable operations, so I wouldn't expect changing the condition to `x == x` to necessarily solve this issue.

But we could replace the floating-point arithmetic with something more reliable, like integer arithmetic; the test `x == x` *will* always succeed if `x` is an integer. Then, Java is complaining about an "error" that is *guaranteed* not to happen.

When such issues are raised with advocates of typing (and in this situation they will often be called advocates of *static typing*, because their opponents claim to support "types", as long as the "types" are only checked dynamically), they might respond like this:

- (Appeal to dogma) You shouldn't write an else branch without a `return` anyway. Types are like logic! Types are part of the fabric of the universe! And *of course* a function should always return something. (Unless it runs forever. Or raises an exception. Both of which are not terribly logical...)[2]
- (Appeal to courtesy) You shouldn't do that, because someone else (such as yourself, a year later) should be able to read your function and understand immediately that it will return something. If they need to reason about the `if` condition to understand why the function will return something, that's not immediate.
- (Appeal to simplicity) Maybe the language should let you do that, but it would make the type system harder to understand, and the type checker harder to implement.

[2] To be fair, many of the people who are most dogmatic about this are also interested in advanced type systems where you can prove that your program won't run forever.

7.8 noreturn examples

7.8.1 noreturn.c

```
#include <stdio.h>

int f (int x)
{
  x * 2;
}

int main (int argc, char **argv)
{
  printf ("noreturn returned: %d\n", f(5));
}
```

With `gcc`: no warnings.
With `gcc -Wall`:

```
noreturn.c: In function 'f':
noreturn.c:5: warning: statement with no effect
noreturn.c:6: warning: control reaches end of non-void function
```

7.8.2 noreturn.java

```
public class noreturn {

  public static int f (double x) {
    if (x == (x + 0.1 - 0.1)) {
      return 0;
    } else {
      // return 1;     // even when there is a return in the then-branch,
                       // Java rejects this program because
                       // there's no return in the else-branch.
    }
  }

  public static void main(String[] args) {
    System.out.println(f (5.0));
  }

}
```

7.9 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!

7.10 Code

The code for these notes can be found in

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

7.11 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.

7.12 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.

7.13 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!

7.13.1 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 $(\mathsf{Lam}\ x\ e)$, 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.

Typing contexts	Γ	$::= \emptyset$	empty context (no assumptions)
		$\mid x : A, \Gamma$	x has type A, with more assumptions

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.

7.13.2 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 7.1. 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 7.1 Typing rules for AE expressions

7.13.3 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

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

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

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

Figure 7.2 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 20. Try to derive the above judgment.

Exercise 21. Derive $y : \mathsf{num} \vdash (\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ x)) : \mathsf{num}$.

Exercise 22. 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.

7.13.4 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 ⟨E⟩) 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.

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

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

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

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

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

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

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

Figure 7.3 Typing rules for expressions for AEL+booleans

■ **Exercise 23.** 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).

7.13.5 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 $\mathsf{op} : A1 * A2 \to B$, 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

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

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

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

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

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

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

Figure 7.4 Typing rules for Typed Fun... not implementable!

7.13.5.1 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).

7.13.5.2 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.[3])

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

7.14 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.

7.15 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.

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

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.

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

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}}}}}}}
```

8 Recap; strings

8.1 Review

Defining programming languages:

- Defining syntax: BNF
- Defining semantics: rules

8.1.1 BNFs

Here's a BNF:

Characters	⟨ch⟩	::=	a | b | c | ···	
Strings	⟨S⟩	::=	"⟨ch⟩ ... "	⟨ch⟩ ... means zero or more repetitions of ⟨ch⟩
Cats	⟨C⟩	::=	⟨S⟩	
			| {+ ⟨C⟩ ⟨C⟩}	

Read "::=" as "can have the form". Other readings you may come across are "expands to" or "rewrites to", which are reasonable in some contexts, but I feel they're misleading in the context of programming languages. We are given an input string (a program) and want to parse it; if there is "rewriting" happening (and I'm not sure there is), it should be going from right to left: the parser sees `b` and realizes it is a character ⟨ch⟩.

Symbols on the **left** of the "::=", like ⟨ch⟩, ⟨S⟩, ⟨C⟩, are called *nonterminals*. On the right hand side, alternatives separated by "|" are called *productions*.

In a BNF, when a nonterminal appears twice, it can (and usually does) represent a different string. For example,

```
{+ "ab" "cd"}
```

is a ⟨C⟩ because `"ab"` and `"cd"` are each ⟨C⟩s (because they are each an ⟨S⟩ (because ...)).

By itself, a BNF only tells you what input strings (programs) are syntactically valid. You might be able to *guess* that I want to define a simple language of string concatenation, where you can "run" `{+ "ab" "cd"}` and get `"abcd"`, but the BNF doesn't say that.

Remark. Unlike "formal semantics" (rules), "formal syntax" (BNF) is used for most "real" programming languages, so it's important to understand it. You also have to be prepared for variations in notation (which is why I try to be careful to always tell you what "..." means).

8.1.2 Abstract syntax

Lisp was supposed to have a "real" syntax, which was never finished. But this piece of vaporware led to something useful: abstract syntax.

Unlike most "real" syntaxes, abstract syntax is not ambiguous; it doesn't need to resolve the ambiguity of `a + b * c`, because everything is in brackets/parentheses/braces.

The **define-type** feature of Racket/PLAI is ideally suited to defining abstract syntax: everything is in parentheses, because everything in Racket is in parentheses.

```
(define-type Cat
   [c/string (s string?)]
   [c/concat (left Cat?) (right Cat?)]
)
```

In the concrete syntax BNF, I could just write ⟨S⟩ by itself as a production of ⟨C⟩, In abstract syntax, each alternative has to begin with a variant name (like `c/string`).

Here, I have written `c/concat` instead of `+`, but this is still only syntax. Using the name `c/concat` strongly suggests that this is meant to be string concatenation, but so far, that's only a name.

8.1.3 Rules

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_m}{\text{conclusion}}\ \text{rule name}$$

There might be no premises. We've seen that with rules like

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num}$$

The conclusion is always some *judgment*, like $e \Downarrow v$ or $\Gamma \vdash e : A$. The conclusion and premises usually have *meta-variables*, which we can replace with instances.

To apply a rule, you replace all its meta-variables. Eval-num has one meta-variable, n. If I instantiate it with 4, I get a *derivation* of $(\mathsf{Num}\ 4) \Downarrow (\mathsf{Num}\ 4)$.

$$\frac{}{(\mathsf{Num}\ 4) \Downarrow (\mathsf{Num}\ 4)}\ \text{Eval-num}$$

In a rule, unlike a BNF, repeated occurrences of the same meta-variable refer to the same thing. So you can't replace the first n with 4, and the second n with 5. (This is a confusing difference, but it's now too standard to change.)

s ::= Racket string

Cat c ::= (c/string s)
  | (c/concat c c)

v ::= (c/string s)

$c \Downarrow v$ Cat c evaluates to v

$$\frac{}{(\text{c/string } s) \Downarrow (\text{c/string } s)} \qquad \frac{}{v \Downarrow v}$$

$$\frac{c_1 \Downarrow (\text{c/string } s_1) \quad c_2 \Downarrow (\text{c/string } s_2) \quad s = s_1 s_2 \ [s_1 \neq s_2]}{(\text{c/concat } c_1\ c_2) \Downarrow (\text{c/string } s)}$$

(concatenation; $s_1 s_2$)

$$\frac{c_1 \Downarrow c \quad c = (\text{c/string } s_1)}{(\text{c/concat } c_1\ c_2) \Downarrow (\text{c/string } \quad)}$$

$$\frac{\dfrac{(\text{c/string "a"}) \Downarrow (\text{c/string "a"}) \quad (\text{c/string "b"}) \Downarrow (\text{c/string "b"})}{(\text{c/concat } (\text{c/string "a"})\ (\text{c/string "b"})) \Downarrow (\text{c/string "ab"})} \quad (\text{c/string "d"}) \Downarrow (\text{c/string "d"})}{(\text{c/concat } \underbrace{(\text{c/concat } (\text{c/string "a"})\ (\text{c/string "b"}))}_{c_1}\ \underbrace{(\text{c/string "d"})}_{c_2}) \Downarrow (\text{c/string "abd"})}$$

8.2 Assignment 3: lists

`List` *A* is the type of lists whose elements are of type *A*. Not like a Racket list, where a list is an arbitrary sequence of stuff.

Expanding on the terse remark that `list-case` is "is a kind of **type-case** for lists":

```
(define-type List-Num
  [numlist-empty ()]
  [numlist-cons (head number?) (tail List-Num?)])

(type-case List-Num xs
  [numlist-empty ()     branch for when xs is numlist-empty]
  [numlist-cons (h t)   branch for when xs is numlist-cons])

{list-case xs {empty => branch for when xs is empty}
              {cons h t => branch for when xs is cons}}
```

Within the `numlist-cons` branch of the Racket/PLAI **type-case**, the identifiers `h` and `t` are bound to the first and second arguments of `numlist-cons`. Similarly, within the `cons` branch of the Fun `list-case`, the identifiers `h` and `t` are bound to the head (first element) and tail (remaining elements) of the list `xs`.

8.2.1 A useful way to read typing rules

The next page illustrates how to "expand" typing rules so you can implement them more directly. When you make a recursive call to derive a premise, you can't constrain in advance what result you get. If you write the rule a little differently, you get something that matches the code you write more closely.

$$\frac{\Gamma \vdash e : A1 \quad A1 = \text{List } A \quad \Gamma \vdash eEmpty : B \quad xh : A, xt : \text{List } A, \Gamma \vdash eCons : B2 \quad B2 = B}{\Gamma \vdash (\text{list-case } e\ eEmpty\ xh\ xt\ eCons) : \begin{matrix} B \\ B2 \end{matrix}}$$

$$\frac{\Gamma \vdash e : A \quad A = A1 * A2 \quad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\text{par-case } e\ x1\ x2\ eBody) : B} \quad (\text{t/* A1 A2})$$

$$\frac{u : B, \Gamma \vdash e : B2 \quad B2 = B}{\Gamma \vdash (\text{rec } u\ B\ e) : \begin{matrix} B \\ B2 \end{matrix}}$$

arguments to type of inputs — result of type of output

$$\Gamma \vdash e : A$$

You can't pass an input to an output...
but you can get the output and then call
type =? or do a type-case.

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash eThen : A \quad \Gamma \vdash eElse : A}{\Gamma \vdash (\text{ite } e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B1 \quad B1 = \text{Bool} \quad \Gamma \vdash eThen : B2 \quad \Gamma \vdash eElse : B3 \quad B2 = B3}{\Gamma \vdash (\text{ite } e\ eThen\ eElse) : B3}$$

$$\frac{\Gamma \vdash e : A \quad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\text{with } x\ e\ eBody) : B}\ \text{Type-with}$$

$$\frac{\Gamma \vdash e : A \quad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\text{with } x\ e\ eBody) : B}$$

8.3 Strings, continued

8.3.1 BNFs

Strings ⟨S⟩ ::= *whatever a Racket string is*

Cats ⟨C⟩ ::= ⟨S⟩
| {+ ⟨C⟩ ⟨C⟩}

Instead of studying the above (very small!) language, we'll add its features to one of our versions of Fun, so that we can see, in a slightly more realistic language, how to define evaluation and typing for these features.

Expressions ⟨E⟩ ::= ...*whatever is in* `typed-lam.rkt`
| ⟨S⟩
| ~~{cat ⟨C⟩ ⟨C⟩}~~

| {cat ⟨E⟩ ⟨E⟩}
| {nth ⟨E⟩ ⟨E⟩}

I've crossed out one of the productions, because I want strings to be interoperable with Fun expressions, so that we can `cat` two identifiers, or `cat` the result of applying two functions, etc.

What is the semantics of `nth`? It will return the 1-character string at a given index into the string, which will illustrate some language design alternatives.

8.3.2 Abstract syntax

```
(define-type E
   .
   .
   .
   [str (s string?)]
   [cat (str1 E?) (str2 E?)]
   [nth (str E?) (index E?)]
)
```

8.3.3 Evaluation rules

$\boxed{e \Downarrow v}$ Expression e evaluates to value v

$$\frac{}{(\mathsf{str}\ s) \Downarrow (\mathsf{str}\ s)}\ \text{Eval-str} \qquad \frac{e1 \Downarrow (\mathsf{str}\ s1) \qquad e2 \Downarrow (\mathsf{str}\ s2)}{(\mathsf{cat}\ e1\ e2) \Downarrow (\mathsf{str}\ s1\ s2)}\ \text{Eval-cat}$$

$$\frac{eS \Downarrow (\mathsf{str}\ s1) \qquad eIdx \Downarrow (\mathsf{num}\ n) \qquad n \in \mathbb{N} \qquad n < \mathit{len}(s1)}{(\mathsf{nth}\ eS\ eIdx) \Downarrow (\mathsf{str}\ s1_n)}\ \text{Eval-nth}$$

The difference between the string s1 and (str s1) is that s1 is a sequence of characters, for which we can define (or assume) various mathematical functions, while (str s1) is abstract syntax.

In writing the rule Eval-nth, we assumed that $\mathbb{N}$ are the natural numbers (and that they start at 0, which is the usual convention in computer science but not necessarily other fields), that $\mathit{len}(s)$ is a (mathematical) function that returns the number of characters in s, and that a subscript like

$$s1_n$$

denotes the nth character of the string s1.

We arrived at the third and fourth premises of Eval-nth by something like the following process.

- First, we voted overwhelmingly (apparently influenced by the federal election) that strings should be indexed from 0 rather than 1.
- Second, we decided (less democratically) to require n to be an integer, rather than taking the floor $\lfloor n \rfloor$. (Because Fun's numbers are the same as Racket's numbers, a num in Fun can be floating-point, rational, or even complex.)
- Third, we decided that n should be required to be in the range $0 \le n < \mathit{len}(s1)$, rejecting a suggestion that we define it "circularly" by taking n mod $\mathit{len}(s1)$.

  (Another possible suggestion: "pin" n to the range, by returning the 0th character when $n < 0$, and the last character when $n \ge \mathit{len}(s1)$. Both this suggestion and the "circular" suggestion don't entirely succeed in their questionable goal of always returning *something*: what should evaluation do if the string's length is zero?)

8.3.4 Errors

What if eIdx evaluates to something that isn't a num?
What if eS evaluates to something that isn't a str?
Both of these can be easily prevented using types.

What if eIdx does evaluate to (num n), but n is not an integer? This is feasible to prevent using types, say, by removing the type num and putting in int and Float types instead, but we won't pursue that now.

What if n falls outside the string? This is much more difficult to prevent with a type system, but it *is* possible. (During the lecture, an abbreviated and questionable attempt to explain how to do this occurred.)

8.3.5 "Going wrong"

A slogan of types advocates is: "Well-typed programs don't go wrong."

This slogan only makes sense if we specifically define what "wrong" means. Then, there are *particular kinds of errors* that are prevented by the typing rules.

The slogan comes from a paper by Robin Milner (the main inventor of Standard ML), who—in his defence—*did* precisely define what he thought "wrong" meant: essentially, it prevented "agreement errors" like trying to apply a number (that is, to call a number as if it were a function), or passing a list to a function that expects an integer, and so on. As you all know by now, such errors happen fairly often, so there's a strong argument for preventing them.

8.4 Typing rules

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

$$\frac{}{\Gamma \vdash (\mathsf{str}\ s) : \mathsf{string}}\ \text{Type-str} \qquad \frac{\Gamma \vdash e1 : \mathsf{string} \qquad \Gamma \vdash e2 : \mathsf{string}}{\Gamma \vdash (\mathsf{cat}\ e1\ e2) : \mathsf{string}}\ \text{Type-cat}$$

$$\frac{\Gamma \vdash eS : \mathsf{string} \qquad \Gamma \vdash eIdx : \mathsf{num}}{\Gamma \vdash (\mathsf{nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

"Expanding" the above rules as discussed above gives:

$$\frac{}{\Gamma \vdash (\mathsf{str}\ s) : \mathsf{string}}\ \text{Type-str} \quad \frac{\Gamma \vdash e1 : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash e2 : A2 \qquad A2 = \mathsf{string}}{\Gamma \vdash (\mathsf{cat}\ e1\ e2) : \mathsf{string}}\ \text{Type-cat}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \mathsf{num}}{\Gamma \vdash (\mathsf{nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

8.5 Type safety

The standard way of showing that a type system really prevents (certain kinds of) errors is to prove *type safety*.

Type safety is a result about the *relationship* between the static semantics and the dynamic semantics. Thus, changing either set of rules can break type safety.

Type safety is more usefully stated for a small-step semantics ($e1 \longrightarrow e2$) rather than for a big-step evaluation semantics, but you're more familiar with the big-step semantics, so we'll start with that.

Type safety can be divided into two parts: *preservation* and *progress*.

8.5.1 Preservation

Preservation says, roughly, that evaluation "preserves types": if you run a program of type bool, and it evaluates to a value, that value will also have type bool.

For the **big-step semantics** $e \Downarrow v$, preservation can be stated as:

If $\emptyset \vdash e : A$
and $e \Downarrow v$
then $\emptyset \vdash v : A$.

Preservation is a limited statement that can best be characterized as: "If you got a value, *then* it is a reasonable value." For example, preservation tells you that if the typing rules say that the expression (App (Lam x num x) (Num 3)) has type num, you won't somehow get a bool instead.

The above preservation statement is actually even more limited than it might appear: if evaluation loops infinitely due to a rec, the above preservation result doesn't help us, because we can only apply it if $e \Downarrow v$ holds.

Nonetheless, preservation should still tell us that some, maybe all, of the errors that `interp` can raise will never happen. (You should be skeptical of even this claim! What could go wrong?)

For the **small-step semantics** $e1 \longrightarrow e2$, preservation can be stated as:

If $\emptyset \vdash e1 : A$
and $e1 \longrightarrow e2$
then $\emptyset \vdash e2 : A$.

8.5.2 Progress

For the small-step semantics, **progress** can be stated as:

If $\emptyset \vdash e1 : A$ then **either**

- $e1$ is a value, or
- $e1 \longrightarrow e2$.

For a big-step semantics $e \Downarrow v$, there is (for most languages) no directly corresponding progress result. The following doesn't hold for Typed Fun, for example, because of rec.

If $\emptyset \vdash e : A$
then $e \Downarrow v$.

A key benefit of small-step semantics is that preservation and progress tell us that running a program, even one that loops infinitely, won't launch the missiles along the way.

9 Polymorphism

9.1 What is polymorphism?

In a language with polymorphism (*poly* = many; *morph* = form), some features of the language can operate with *multiple types*. "Some features" and "can operate with" are deliberately vague: there are many kinds of polymorphism, and a given language might allow one kind for some language fatures, under some circumstances, and another kind of polymorphism in others.

9.2 Kinds of polymorphism

In 1967, Christopher Strachey (who made important contributions to programming language semantics, *and* designed a key ancestor of C) distinguished two kinds of polymorphism:

- parametric polymorphism, and
- *ad hoc* polymorphism.

A further kind of polymorphism, perhaps the kind you've used the most, is *subtype polymorphism*, also called *inclusion polymorphism*. For example, if you have a pair of type $\mathsf{pos} * \mathsf{pos}$, you should be able to pass it to a function of type $(\mathsf{rat} * \mathsf{rat}) \to \mathsf{bool}$.

9.2.1 Examples of parametric polymorphism

In parametric polymorphism, types include *type variables* that can be *instantiated*.

(see poly.sml)

To understand these types, we should really write the *quantifiers* that SML (implicitly) puts around these types. For example, `identity_function` has type

$$\forall\alpha.\ (\alpha \to \alpha) \qquad \text{"for all types } \alpha, \ldots\text{"}$$

That is, any code that calls `identity_function` can provide something of any type it chooses, and will (if evaluation results in a value!) get back something of that same type.

```
identity_function 5;
identity_function (1, 2);
```

In the first line above, `5` has SML type `int`, so SML *instantiates* α with `int`, resulting in the type

$$(\texttt{int} \to \texttt{int})$$

Applying a function of type (`int` → `int`) to an `int` results in an `int`, so `identity_function 5` has type `int`.

A larger example is `map_list`, which has the polymorphic type

$$\forall\alpha.\ \big(\forall\beta.\ (\alpha \to \beta) \to (\alpha\ \texttt{list}) \to (\beta\ \texttt{list})\big)$$

This type says: if you pick types α and β (which, like meta-variables in typing rules, might or might not be *different* types), and pass (first) a function of type $\alpha \to \beta$ and (second) a list whose elements all have type α, then the value returned by calling `map_list` (if that call returns at all) will be a list whose elements are of type β.

(illustrate with `map_list make_pair` from poly.sml)

The reason this is called *parametric* polymorphism is that the types α and β don't matter: the implementation of `map_list` doesn't care what types you instantiate α and β with. In fact, in SML it is *impossible* for `map_list` to know which types α and β have been instantiated with!

If you try to do something that depends on α having a particular type, SML will infer a "less polymorphic" type instead:

```
val unpoly_map_list = fn : (bool -> 'b) -> bool list -> 'b list
```

The fact that a parametrically polymorphic function *cannot* inspect its argument's type means that we can prove "parametricity properties", such as:

> If a function has type $\forall\alpha.\ (\alpha \to \alpha)$, and it is applied to a value v of some type A, and that application evaluates to a value, then the resulting value *is exactly* v.

Or, suppose a function has type $\forall\alpha.\ \big((\alpha * \alpha) \to \alpha\big)$. It could return the first part of the pair, or the second part. Could it do anything else?

Turning the question around (sideways?): What functions *besides* `map_list` have `map_list`'s type?

9.2.2 Examples of ad hoc polymorphism

A common form of *ad hoc* polymorphism is *operator overloading*: in many languages, the `+` operator works on more than one type of argument. For example, in SML, `+` works on both `int`s and `real`s (though not on `string`, and not on one `int` and one `real`).

9.2.3 Polymorphism in untyped languages

Is Racket polymorphic? The answer depends on whether we take "type" in the (vague) definition above to mean a static type (perhaps defined through typing rules), or whether we consider it more informally, so that, say, `3` and `#false` in Racket are of different types, even though Racket has no type system to stop you from compiling a program like `(+ 3 #false)`.

- If we require "type" to mean a static type, then Racket is not polymorphic because, in a sense, it has *only one type*: the type of "s-expressions", which includes numbers, `#true` and `#false`, functions (`lambda`), lists, and everything else.

  This claim is sometimes phrased as "dynamic 'typing' is *really* just *unityping*", a "unityped" language being a (statically) typed language with only one (*uni-*) type. Thus, Carnegie Mellon University's Bob Harper:

  > "Dynamic typing is but a special case of static typing, one that limits, rather than liberates... Something can hardly be *opposed* to that of which it is but a trivial special case." (from a 2011 blog post)

- If we say that *any* precise organization of code and/or data into subcategories is "typing", then `#true` and `#false` can be called "booleans", (**lambda** `(x) x`) can be called a "function", and so on. Then Racket is certainly polymorphic, because many functions that you can write in Racket—for example, (**lambda** `(x) x`)—work on many different kinds of Racket "types".

10 Environments

10.1 The trouble with substitution

We've defined dynamic semantics in two different ways: big-step and small-step. In both, we used substitution to define what expressions with identifiers (variables) mean: a with expression evaluates its bound expression, and immediately replaces all the instances of the bound identifier with that expression. Functions (lam/app) and recursion (rec) also were given meaning via substitution.

Our notion of substitution is directly descended from Church's λ-calculus, but as a general (and less precise) notion, substitution is older: in algebra we can substitute 5 for x in

$$x^2 + 3$$

to get $5^2 + 3$. (I don't think the ancient syllogisms of Greece and India—"Socrates is a man, all men are mortal, therefore Socrates is mortal"; "This hill is smoky; whatever is smoky is fiery (for example: a kitchen); therefore this hill is fiery"[1]—are truly *substitution*: there are no variables.)

The connection to the λ-calculus, which was shown to be equivalent in power to Turing machines, guarantees that substitution is *a* "right way" of defining how features like with and app work. It does not mean that substitution is *the* right way of defining how those features work. In fact, substitution is (almost?) never used to implement interpreters.

Substitution has several disadvantages, compared to other methods:

- *Inefficiency:* Every time our interpreter calls $[e2/x]e1$, our implementation of *subst* searches for (id x) throughout the entire expression $e1$. It must do this even if $e1$ is very large, and (id x) appears just once (or even not at all!).

- *Obscurity:* Giving a function (or other expression) a name is important for clarity and convenience; we would rather write `{app double 5}` than `{app {lam y {+ y y} 5}}`, even though they give the same result. But a substitution-based interpreter that prints the expressions it's evaluating (like `visible-interp.rkt` does) will show you the latter. This is perhaps most aggravating with recursive functions.

Against these, we should weigh substitution's advantages:

- *Simplicity:* The definition of substitution is more concise and straightforward than other methods.

[1] Adapted from Vidyabhusana, *A History of Indian Logic* (1920), p. 61.

- *Versatility:* While substitution doesn't "scale" in terms of performance (see "Inefficiency" above), it "scales up" well across a variety of language features. The same, relatively simple, style of defining substitution works for languages with functions that return functions ("first-class" functions) and for recursive functions. Environments are more brittle: adding new features sometimes requires us to define environment in a way that is more complicated (rather than just being *longer*, as is the case with substitution).

Whether or not you prefer environments, you should learn about them, especially if you plan to take CPSC 411.

10.2 Environments

The idea of environment-based dynamic semantics is that, to evaluate (Let x *e*2 *e*1), we won't evaluate *e*2 to *v*2 and then substitute *v*2 for x; instead, we will evaluate *e*2 to *v*2, and "remember" the fact that x has the value *v*2. This fact will be stored in an *environment* that maps identifiers to values. If and when we need to evaluate an instance of x in the body *e*1, that is, if we need to evaluate (Id x), we look up x and use the value we find, which will be *v*2.

In a sense, we are simulating substitution: if we had substituted *v*2 for x, we would find *v*2 inside the body *e*1.

10.2.1 Back to basics: WAE

Because environments are more brittle than substitution, I think it's better to "roll back" our Fun language to WAE (arithmetic expressions and with), define the simplest possible environments, and then carefully evolve our notion of an environment as we restore language features.

Quoting `04-operational.pdf`:

```
⟨WAE⟩ ::= ⟨num⟩
        | {+ ⟨WAE⟩ ⟨WAE⟩}
        | {- ⟨WAE⟩ ⟨WAE⟩}
        | {with {⟨id⟩ ⟨WAE⟩} ⟨WAE⟩}
        | ⟨id⟩
```

```
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)])
```

10.2.2 Mapping identifiers to expressions

To get an idea of what is needed, consider the WAE expression (in abstract syntax)

$$(\mathsf{Let}\ x\ (\mathsf{Num}\ 3)\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y))))$$

If we don't use substitution, when we evaluate $(\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y))$ we need to remember that x was bound to $(\mathsf{Num}\ 3)$, and y was bound to $(\mathsf{Num}\ 4)$. We need a "lookup table" that maps identifiers to expressions.

In Typed Fun, we used a typing context Γ to map identifiers to types, and defined what those contexts were with a grammar:

Typing contexts	Γ	$::= \emptyset$	empty context (no assumptions)
		$\mid x : A, \Gamma$	x has type A, with more assumptions

We'll do the same for environments:

Environments (for WAE)	env	$::= \emptyset$	empty environment
		$\mid x{=}e, env$	x bound to e, with "more environment"

(It would be more standard to use the Greek letter rho (ρ), rather than "env", but we've used enough Greek letters for now.)

For consistency with typing contexts Γ, environments env will grow to the left, like `cons`-lists in Racket.

Let's consider an even smaller example than the one above.

$$(\mathsf{Let}\ y\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y)))$$

If this expression is the entire program, it's not inside any withs, so the environment env is empty when we start evaluating.

Regardless of how environments work, $(\mathsf{Num}\ 4)$ should still evaluate to $(\mathsf{Num}\ 4)$. But now we need to remember that y is $(\mathsf{Num}\ 4)$, so we'll need to evaluate the body $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y))$ under the environment

$$y{=}(\mathsf{Num}\ 4), \emptyset$$

(It's okay to write this as just $y{=}(\mathsf{Num}\ 4)$; here, I want to emphasize that we started with $\emptyset$, and are growing the environment leftwards.)

Then, while evaluating $(\mathsf{Id}\ y)$ in $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y))$, we will look up $(\mathsf{Id}\ y)$ in the current environment $y{=}(\mathsf{Num}\ 4), \emptyset$, and evaluation will behave as if we were evaluating $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4))$.

Just as we used Γ in the typing judgment $\Gamma \vdash e : A$, we'll use env in a new *environment-based evaluation* judgment form

$$env \vdash e \Downarrow v$$

We'll also assume that a "lookup function" $lookup(env, x)$ has been defined, so that

$$lookup(env, x) = e$$

if the environment env contains $x{=}e$. (In our Racket code, we have a function `look-up-id`.)

I think we have enough to revise the evaluation rules for WAE. What were those?

$\boxed{e \Downarrow v}$ WAE expression e evaluates to value v

$$\frac{}{(\text{num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Eval-add}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Eval-with}$$

$$\frac{}{(\text{id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, WAE expression e evaluates to value v

$$\frac{}{env \vdash (\text{num } n) \Downarrow (\text{Num } n)}\ \text{Env-num}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Env-add}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Env-sub}$$

$$\frac{e1 \Downarrow v1 \qquad \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Env-with}$$

$$\frac{}{(\text{Id } x) \Downarrow}\ \text{Env-id}$$

$$\frac{}{\text{unknown-id-error}}\ \text{Env-unknown-id}$$

Exercise 24. (Do it tonight, before class, if feasible.)
I left some blank space in the "Env-..." rules. Fill it in with whatever is needed. Env-num is finished, and you can follow that pattern for some of the other rules.

If you're not sure how to start, I already updated *part* of the function `env-interp` in `env-with-broken.rkt` (link on the notes page) to reflect how I would fill in Env-add and Env-sub, so you can map back from that code if you like. But I haven't written the code for the more interesting rules yet...

The completed rules are on the next page.

$\boxed{e \Downarrow v}$ WAE expression e evaluates to value v

$$\frac{}{(\text{num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Eval-add}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Eval-with}$$

$$\frac{}{(\text{id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, WAE expression e evaluates to value v

$$\frac{}{env \vdash (\text{num } n) \Downarrow (\text{Num } n)}\ \text{Env-num}$$

$$\frac{env \vdash e1 \Downarrow (\text{Num } n1) \qquad env \vdash e2 \Downarrow (\text{Num } n2)}{env \vdash (\text{add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Env-add}$$

$$\frac{env \vdash e1 \Downarrow (\text{Num } n1) \qquad env \vdash e2 \Downarrow (\text{Num } n2)}{env \vdash (\text{sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Env-sub}$$

$$\frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Env-with}$$

$$\frac{lookup(env, x) = e}{env \vdash (\text{Id } x) \Downarrow e}\ \text{Env-id}$$

$$\frac{lookup(env, x)\ \text{undefined}}{env \vdash (\text{Id } x)\ \text{unknown-id-error}}\ \text{Env-unknown-id}$$

10.2.3 The Shadow Chancellor Strikes Back

(At some point, the UK Parliament becomes indistinguishable from a bad fantasy novel.)

With substitution, we saw that expressions that repeatedly bind the same identifier are evaluated with the inner binding "shadowing" the outer one, so that

$$(\text{Let } x\ (\text{Num } 1)\ (\text{Let } x\ (\text{Num } 2)\ (\text{Id } x)))$$

evaluates to (Num 2), not (Num 1). The environment-based semantics will behave the same way, but only because of a particular way we're defining *lookup*: it starts looking from the left.

$$\dfrac{\dfrac{}{\emptyset \vdash (\text{Num } 1) \Downarrow (\text{Num } 1)}\ \text{Env-num} \qquad \dfrac{\dfrac{}{x{=}(\text{Num } 1), \emptyset \vdash (\text{Num } 2) \Downarrow (\text{Num } 2)}\ \text{Env-num} \qquad \dfrac{lookup((x{=}(\text{Num } 2), x{=}(\text{Num } 1), \emptyset), x) = (\text{Num } 2)}{x{=}(\text{Num } 2), x{=}(\text{Num } 1), \emptyset \vdash (\text{Id } x) \Downarrow (\text{Num } 2)}\ \text{Env-id}}{x{=}(\text{Num } 1), \emptyset \vdash (\text{Let } x\ (\text{Num } 2)\ (\text{Id } x)) \Downarrow (\text{Num } 2)}\ \text{Env-with}}{\emptyset \vdash (\text{Let } x\ (\text{Num } 1)\ (\text{Let } x\ (\text{Num } 2)\ (\text{Id } x))) \Downarrow (\text{Num } 2)}\ \text{Env-with}$$

We should really define *lookup* using rules; I'll leave that as an exercise (next page).

Exercise 25. Fill in the rules below, which derive a judgment $lookup(env, x) = e$:
(Feel free to translate "backwards" from the Racket implementation of `look-up-id`.)

$$\frac{}{lookup(\emptyset, x) = \_\_\_}\ \text{lookup-empty}$$

$$\frac{}{lookup((x{=}e, env), x) = \_\_\_}\ \text{lookup-found} \qquad \frac{}{lookup((y{=}e, env), x) = \_\_\_}\ \text{lookup-next}$$

10.2.4 Question Period

Question:
The expression after the "⇓" should always be a value. Shouldn't Env-id evaluate e to v?

It could, but it doesn't need to: the expressions we put into environments are all values. The only rule that adds anything to the environment is Env-with, and the expression it adds is $v1$, which is a value.

Question: Could Env-with *not* evaluate $e1$, and put $e1$ into the environment, instead?

In that case, Env-id *would* need to evaluate the expression it gets from *lookup*. That would give us an "expression strategy" for with. That's inconsistent with our substitution-based semantics, but it's not wrong; it's just not what I want to do.

Question: What if Env-with puts $e1$ into the environment, and Env-id evaluates that expression to get $v1$, and then *updates the environment* with $v1$? Would that gives us lazy evaluation?

You could certainly implement that—for example, using Racket's mutable "boxes". Moreover, we could model it using rules. But the rules would need to be rather different from the above rules, which derive the judgment form $env \vdash e \Downarrow v$. That judgment can't model a change to the environment; the above rules can only add to the environment *inside* a premise. So if your environment is

$$\underbrace{\mathsf{x}{=}(\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1)), \emptyset}_{env}$$

and you evaluate $(\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))$, you can't "transmit" the updated env from the first premise to the second premise. Rules and derivations aren't mutable.

$$\frac{env \vdash (\mathsf{Id}\ \mathsf{x}) \Downarrow (\mathsf{Num}\ 2) \qquad env \vdash (\mathsf{Id}\ \mathsf{x}) \Downarrow (\mathsf{Num}\ 2)}{\underbrace{\mathsf{x}{=}(\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1)), \emptyset}_{env} \vdash (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})) \Downarrow (\mathsf{Num}\ 4)}\ \text{Env-add}$$

However, you could change the judgment form to something like

$$env \vdash e \Downarrow v, env'$$

which could be read "starting in environment env, evaluating expression e produces value v and an environment env'." Then the conclusion of the rule for id could have the "updated" environment as env'.

We'll need to do something like this to model mutable state (hopefully, next week).

11 Closures

11.1 Attack of the Dynamic Scope

Fun is broken! (Typed Fun is *not* broken, though! Types win again?)

Scoping in Fun was supposed to be *lexical*, where an instance of an identifier refers to the nearest enclosing binding occurrence. Thus, in

$$(\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x)))$$

the instance $(\mathsf{Id}\ x)$ refers to the inner binding occurrence (and therefore to $(\mathsf{Num}\ 2)$).

Before we added Lam and App, scoping actually *was* properly lexical: attempting to evaluate an identifier that has *no* enclosing binder, as in

$$(\mathsf{Let}\ x\ (\mathsf{Id}\ y)\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x)))$$

would cause a free variable error.

However, scoping in Fun was partly lexical, and partly *dynamic*:

$$\Big(\mathsf{Let}\ f\ \big(\mathsf{Lam}\ y\ (\mathsf{Id}\ z)\big)\ \big(\mathsf{Let}\ z\ (\mathsf{Num}\ 2)\ (\mathsf{App}\ (\mathsf{Id}\ f)\ (\mathsf{Num}\ 0))\big)\Big)$$

The rule Eval-let substitutes $(\mathsf{Lam}\ y\ (\mathsf{Id}\ z))$ for f, and then evaluates

$$\big(\mathsf{Let}\ z\ (\mathsf{Num}\ 2)\ (\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Id}\ z))\ (\mathsf{Num}\ 0))\big)$$

in which $(\mathsf{Num}\ 2)$ is substituted for z:

$$(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Num}\ 2))\ (\mathsf{Num}\ 0))$$

which evaluates to $(\mathsf{Num}\ 2)$. Observe that $(\mathsf{Lam}\ y\ (\mathsf{Id}\ z))$ doesn't look at its argument y, which in any case is substituted with $(\mathsf{Num}\ 0)$, which is not $(\mathsf{Num}\ 2)$. In PL jargon, we say that the identifier $(\mathsf{Id}\ z)$ in the body of $(\mathsf{Lam}\ y\ (\mathsf{Id}\ z))$—which really shouldn't refer to *anything* and should be an error—has been *captured* by the binding $(\mathsf{Let}\ z\ (\mathsf{Num}\ 2)\ \ldots)$.

11.1.1 A Brief History of Infamy

The story goes that dynamic scope—in which the "most recent" binding is used, rather than the lexically *enclosing* binding—was invented, by accident, in Lisp. Subsequent versions of Lisp corrected this, except for Emacs Lisp (which most of Emacs is written in).

In Fun, we implemented something that "is" lexical scoping, in the sense that an identifier in any correctly lexically-scoped expression will refer to its nearest lexically-enclosing binder. But we also implemented a little dynamic scoping: in an expression with a `Lam`, we allow free identifiers inside the body, which can then be captured by later bindings.

We corrected this (unknowingly) in Typed Fun: When `typeof` sees an identifier, it checks that the identifier appears in the typing context `tc` (written Γ in the rules).

The fix in Fun itself is—I'm pretty sure—to add a check in `subst` that makes sure the expression being substituted has no free identifiers (*not* the expression being substituted *into*, which probably does have a free identifier: the identifier being substituted!).

For example, in the example with `f` and `(Lam y (Id z))` above, that check would find the free identifier `(Id z)`, and raise an error.

11.2 Functions in environment-based semantics

The eruption of dynamic scoping is relevant, however, because another way to accidentally get dynamic scoping is to add functions to an environment semantics. So let's do that, and then fix it.

$\boxed{e \Downarrow v}$ Fun expression e evaluates to value v

$$\frac{}{(\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{Add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Eval-add}$$

$$\frac{e1 \Downarrow (\text{Num } n1) \qquad e2 \Downarrow (\text{Num } n2)}{(\text{Sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{Let } x\ e1\ e2) \Downarrow v2}\ \text{Eval-let}$$

$$\frac{}{(\text{Id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\text{Lam } x\ e1) \Downarrow (\text{Lam } x\ e1)}\ \text{Eval-lam}$$

$$\frac{e1 \Downarrow (\text{Lam } x\ eB) \quad e2 \Downarrow v2 \quad [v2/x]eB \Downarrow v}{(\text{App } e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, Fun expression e evaluates to value v

$$\frac{}{env \vdash (\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Env-num}$$

$$\frac{env \vdash e1 \Downarrow (\text{Num } n1) \qquad env \vdash e2 \Downarrow (\text{Num } n2)}{env \vdash (\text{Add } e1\ e2) \Downarrow (\text{Num } n1 + n2)}\ \text{Env-add}$$

$$\frac{env \vdash e1 \Downarrow (\text{Num } n1) \qquad env \vdash e2 \Downarrow (\text{Num } n2)}{env \vdash (\text{Sub } e1\ e2) \Downarrow (\text{Num } n1 - n2)}\ \text{Env-sub}$$

$$\frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\text{Let } x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{lookup(env, x) = e}{env \vdash (\text{Id } x) \Downarrow e}\ \text{Env-id}$$

$$\frac{lookup(env, x)\ \text{undefined}}{env \vdash (\text{Id } x)\ \text{unknown-id-error}}\ \text{Env-unknown-id}$$

$$\frac{}{env \vdash (\text{Lam } x\ e1) \Downarrow (\text{Lam } x\ e1)}\ \text{**Env-lam-dynamic}$$

$$\frac{env \vdash e1 \Downarrow (\text{Lam } x\ eB) \quad env \vdash e2 \Downarrow v2 \quad x{=}v2, env \vdash eB \Downarrow v}{env \vdash (\text{App } e1\ e2) \Downarrow v}\ \text{**Env-app-dynamic}$$

11.2.1 Boom! Lambda

(In honour of the failed renaming of Pie R Squared.)

The above rules, which seem reasonable—**Env-app-dynamic follows the pattern of Env-let—cause even more dynamic scoping than my oversight in substitution-based Fun.

Consider the expression

$$\big(\text{Let y (Num 1) (Let f (Lam x (Id y)) (Let y (Num 2) (App (Id f) (Num 0))))}\big)$$

In a substitution-based semantics, the first thing we do is substitute (Num 1) for y:

$$\big(\text{Let f (Lam x (Num 1)) (Let y (Num 2) (App (Id f) (Num 0)))}\big)$$

This means that f (will be substituted with) a constant function that always returns (Num 1).

However, with the above **-rules, we add y=(Num 1) to the empty environment, then f=(Lam x (Id y)), and then y=(Num 2). Since *lookup* looks at the environment starting from the left, looking up an instance of (Id y) will result in (Num 2):

$$\frac{env_{yfy} \vdash \text{(Id f)} \Downarrow \text{(Lam x (Id y))} \qquad env_{yfy} \vdash \text{(Num 0)} \Downarrow \text{(Num 0)} \qquad \text{x=(Num 0)}, env_{yfy} \vdash \text{(Id y)} \Downarrow \text{(Num 2)}}{\underbrace{\text{y=(Num 2)}, \text{f=(Lam x (Id y))}, \text{y=(Num 1)}, \emptyset}_{env_{yfy}} \vdash \text{(App (Id f) (Num 0))} \Downarrow \text{(Num 2)}}\ \text{**Env-app-dynamic}$$

The problem is that when f=(Lam x (Id y)) was added to the environment, looking up (Id y) would have given (Num 1), since that is the nearest enclosing binding. But instead, we used a binding that was nowhere in scope when f was bound.

Under lexical scoping, you can always determine where an identifier's binder is without "looking into the future": if a nested Let that comes later happens to shadow an identifier, it won't matter. Under dynamic scoping, which we have now re-created, this isn't the case.

Question: Can you show the rest of the derivation?

$$\frac{\dfrac{}{\text{(Lam x (Id y))}}}{\text{y=(Num 1)}, \emptyset \vdash \Downarrow \text{(Lam x (Id y))}} \qquad \frac{\dfrac{}{\text{(Num 2)}}}{env_{fy} \vdash \Downarrow \text{(Num 2)}} \qquad \frac{\overbrace{\text{y=(Num 2)}, \text{f=(Lam x (Id y))}, \text{y=(Num 1)}, \emptyset}^{env_{yfy}} \vdash \text{(App (Id f) (Num 0))} \Downarrow \text{(Num 2)}}{\underbrace{\text{f=(Lam x (Id y))}, \text{y=(Num 1)}, \emptyset}_{env_{fy}} \vdash \big(\text{Let y (Num 2) (App } \cdots\text{)}\big) \Downarrow \text{(Num 2)}}\ \text{Env-let}\ \checkmark\ \textit{see derivation above}$$

$$\text{y=(Num 1)}, \emptyset \vdash \big(\text{Let f (Lam x (Id y)) (Let y } \cdots\text{)}\big) \Downarrow \text{(Num 2)}\ \text{Env-let}$$

11.2.2 Closures

The solution is to remember something about the environment that existed *when the binding happened*, that is, when the Lam was evaluated.

The easiest way to remember something about the environment is to remember the entire environment, so that's what we'll do. The "pairing up" of a Lam with its environment is called a *closure*.

Closures are a new kind of animal; where do they live? For uniformity, it will be easiest (I think) to think of them as expressions. Alternatively, we could make them a new kind of thing in the environment, so that we'd have ordinary value bindings in the environment, and also closure bindings.

This leads to the following **define-type**:

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
  [Clo (env Env?) (e E?)]                ; not in concrete syntax
  )
```

We've already seen a **define-type** in which a variant didn't correspond to a single production of the BNF: `Binop`. There, however, the `Binop` variants were generated inside the parser. Here, the parser will never generate a closure `Clo`. Instead, closures will be generated only inside the interpreter `env-interp`.

$$\frac{}{env \vdash (\mathsf{Lam}\ x\ e1) \Downarrow \big(\mathsf{Clo}\ env\ (\mathsf{Lam}\ x\ e1)\big)}\ \text{Env-lam} \qquad \frac{}{env \vdash (\mathsf{Clo}\ env_{old}\ e) \Downarrow (\mathsf{Clo}\ env_{old}\ e)}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{old} \vdash eB \Downarrow v}{env \vdash (\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Env-app}$$

Question: What happens if the Lam has a free variable that isn't in the environment env_{old}, but is in the newer environment we have when we evaluate App? If we need a free-variable check in *subst* for substitution-based semantics, do we also need one for environment-based semantics?

If the Lam tries to use an identifier that isn't in the environment when the Lam was evaluated, this will be an error. The error won't happen until the Lam—which is now a $(\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ldots))$—is applied, but it will be a proper error; the identifier will not be captured by a later binding, because that binding won't be in env_{old}.

If our language is typed, this doesn't matter, because the type checker will catch this error statically.

In practice, particularly in a compiler, we only store the actual free identifiers of the Lam in the closure—not the entire environment. After parsing, we can figure out what the free identifiers of the Lam are, so we'll know which bindings from env_{old} to save.

11.3 Recursive closures

At this point, we could add more features (pairs, trees, sums, etc.) without any trouble. Instead, let's try to add the feature that *will* give us trouble: Rec.

$$
\begin{array}{rrll}
\text{Environments} & env & ::= & \emptyset \\
& & | & x{=}e, env \\
& & | & \mathcal{E} \qquad\qquad\qquad \text{environment variable} \\
& & | & (\text{Env-Rec } \mathcal{E}\ env) \quad \text{recursive environment}
\end{array}
$$

Definition of environment substitution over environments:

$$
\begin{array}{rcl}
[env_1/\mathcal{E}_1]\emptyset & = & \emptyset \\
[env_1/\mathcal{E}_1](x{=}e, env_2) & = & x{=}[env_1/\mathcal{E}_1]e, [env_1/\mathcal{E}_1]env_2 \\
[env_1/\mathcal{E}_1]\mathcal{E}_1 & = & env_1 \\
[env_1/\mathcal{E}_1]\mathcal{E}_2 & = & \mathcal{E}_2 \quad \text{if } \mathcal{E}_1 \neq \mathcal{E}_2 \\
[env_1/\mathcal{E}_1](\text{Env-Rec } \mathcal{E}_1\ env_2) & = & (\text{Env-Rec } \mathcal{E}_1\ env_2) \\
[env_1/\mathcal{E}_1](\text{Env-Rec } \mathcal{E}_2\ env_2) & = & (\text{Env-Rec } \mathcal{E}_2\ [env_1/\mathcal{E}_1]env_2) \quad \text{if } \mathcal{E}_1 \neq \mathcal{E}_2
\end{array}
$$

Definition of environment substitution over expressions:

$$
\begin{array}{rcl}
[env/\mathcal{E}](\text{Num } n) & = & (\text{Num } n) \\
[env/\mathcal{E}](\text{Binop op e1 e2}) & = & (\text{Binop op } [env/\mathcal{E}]\text{e1 } [env/\mathcal{E}]\text{e2}) \\
[env/\mathcal{E}](\text{Btrue}) & = & (\text{Btrue}) \\
[env/\mathcal{E}](\text{Bfalse}) & = & (\text{Bfalse}) \\
[env/\mathcal{E}](\text{Ite e eThen eElse}) & = & (\text{Ite } [env/\mathcal{E}]\text{e } [env/\mathcal{E}]\text{eThen } [env/\mathcal{E}]\text{eElse}) \\
[env/\mathcal{E}](\text{Id } x) & = & (\text{Id } x) \\
[env/\mathcal{E}](\text{Let } x\ \text{e1 e2}) & = & (\text{Let } x\ [env/\mathcal{E}]\text{e1 } [env/\mathcal{E}]\text{e2}) \\
[env/\mathcal{E}](\text{Lam } x\ \text{eB}) & = & (\text{Lam } x\ [env/\mathcal{E}]\text{eB}) \\
[env/\mathcal{E}](\text{App e1 e2}) & = & (\text{App } [env/\mathcal{E}]\text{e1 } [env/\mathcal{E}]\text{e2}) \\
[env/\mathcal{E}](\text{Rec } u\ \text{eB}) & = & (\text{Rec } u\ [env/\mathcal{E}]\text{eB}) \\
[env_1/\mathcal{E}](\text{Clo } env_2\ e) & = & (\text{Clo } [env_1/\mathcal{E}]env_2\ [env_1/\mathcal{E}]e)
\end{array}
$$

Definition of environment lookup:

$$
\begin{array}{rcl}
\text{lookup}\big((x = e, env), x\big) & = & e \\
\text{lookup}\big((y = e, env), x\big) & = & \text{lookup}(env, x) \quad \text{if } x \neq y \\
\text{lookup}\big(\text{Env-Rec } \mathcal{E}\ env), x\big) & = & \text{lookup}\Big(\big[\text{Env-Rec } \mathcal{E}\ env)/\mathcal{E}\big]env, x\Big)
\end{array}
$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, expression e evaluates to value v

$$\frac{}{env \vdash (\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Env-num} \qquad \frac{env \vdash e1 \Downarrow v1 \qquad env \vdash e2 \Downarrow v2 \qquad v1\ op\ v2 = v}{env \vdash (\mathsf{Binop}\ op\ e1\ e2) \Downarrow v}\ \text{Env-binop}$$

$$\frac{}{env \vdash (\mathsf{Btrue}) \Downarrow (\mathsf{Btrue})}\ \text{Env-true} \qquad \frac{}{env \vdash (\mathsf{Bfalse}) \Downarrow (\mathsf{Bfalse})}\ \text{Env-false}$$

$$\frac{env \vdash e \Downarrow (\mathsf{Btrue}) \qquad env \vdash e\mathsf{Then} \Downarrow v}{env \vdash (\mathsf{Ite}\ e\ e\mathsf{Then}\ e\mathsf{Else}) \Downarrow v}\ \text{Env-ite-true}$$

$$\frac{env \vdash e \Downarrow (\mathsf{Bfalse}) \qquad env \vdash e\mathsf{Else} \Downarrow v}{env \vdash (\mathsf{Ite}\ e\ e\mathsf{Then}\ e\mathsf{Else}) \Downarrow v}\ \text{Env-ite-false}$$

$$\frac{lookup(env, x) = e \qquad env \vdash e \Downarrow v}{env \vdash (\mathsf{Id}\ x) \Downarrow v}\ \text{Env-id} \qquad \frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{}{env \vdash (\mathsf{Lam}\ x\ eB) \Downarrow (\mathsf{Clo}\ env\ (\mathsf{Lam}\ x\ eB))}\ \text{Env-lam} \qquad \frac{env_{old} \vdash e \Downarrow v}{env \vdash (\mathsf{Clo}\ env_{old}\ e) \Downarrow v}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{old} \vdash eB \Downarrow v}{env \vdash (\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Env-app-value}$$

$$\frac{\big(\mathsf{Env\text{-}Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ eB), env)\big) \vdash eB \Downarrow v \qquad \mathcal{E} \notin env}{env \vdash (\mathsf{Rec}\ u\ eB) \Downarrow v}\ \text{Env-rec}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{lookup}(\mathit{envrec},u) = \mathit{eLookup}}\quad \dfrac{\dfrac{}{\mathit{lookup}(\mathit{envclo},u) = \quad}\quad \mathit{envclo} \vdash \quad \Downarrow}{\mathit{envclo} \vdash (\mathsf{Id}\ u) \Downarrow}\ \text{Env-id}}{\mathit{envrec} \vdash (\mathsf{Clo}\ \mathit{envclo}\ (\mathsf{Id}\ u)) \Downarrow}\ \text{Env-clo}}{\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big) \vdash (\mathsf{Id}\ u) \Downarrow}\ \text{Env-id}}{\emptyset \vdash \Big(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)\Big) \Downarrow}\ \text{Env-rec}$$

$$\mathit{envrec} = \big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big)$$

$$\begin{aligned}
\mathit{lookup}(\mathit{envrec}, u) &= \mathit{lookup}\Big(\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big),\ u\Big) \\
&= \mathit{lookup}\Big(\Big[\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big) \Big/ \mathcal{E}\Big]\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset),\ u\Big) \\
&= \mathit{lookup}\Big(\big(u{=}(\mathsf{Clo}\ \quad (\mathsf{Id}\ u)), \emptyset),\ u\Big) \\
&= \mathit{lookup}\Big(\big(u{=}(\mathsf{Clo}\ \big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big)\ (\mathsf{Id}\ u)), \emptyset),\ u\Big) \\
&= \mathit{lookup}\Big(\big(u{=}(\mathsf{Clo}\ \big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big)\ (\mathsf{Id}\ u)), \emptyset),\ u\Big) \\
&= \big(\mathsf{Clo}\ \underbrace{\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ (\mathsf{Id}\ u)), \emptyset)\big)}_{\mathit{envclo}}\ (\mathsf{Id}\ u)\big) \\
&= \mathit{eLookup}
\end{aligned}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E} \vdash \mathit{eBody} \Downarrow}{\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ \mathit{eBody}), \emptyset)\big) \vdash (\mathsf{Clo}\ \mathcal{E}\ \mathit{eBody}) \Downarrow}\ \text{Env-clo}}{\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ \mathit{eBody}), \emptyset)\big) \vdash (\mathsf{Id}\ u) \Downarrow}\ \text{Env-id}}{\big(\text{Env-Rec}\ \mathcal{E}\ (u{=}(\mathsf{Clo}\ \mathcal{E}\ \mathit{eBody}), \emptyset)\big) \vdash \big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Id}\ u)\ (\mathsf{Bfalse})\big) \Downarrow}\ \text{Env-ite-true}}{\emptyset \vdash \Big(\mathsf{Rec}\ u\ \underbrace{\big(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Id}\ u)\ (\mathsf{Bfalse})\big)}_{\mathit{eBody}}\Big) \Downarrow}\ \text{Env-rec}$$

The difficulty with Rec is that we need to create a closure in which the saved environment *has a binding to that same closure*. The best way I've found for doing this in Racket is to use *boxes*.

11.3.1 Boxes in Racket

A Racket *box* is like a pointer or reference that points to a value that can be mutated (updated).

- You can create a Racket box with `box`. The way Racket prints a box looks kind of weird. It will get weirder.

```
> (box 5)
'#&5
> (box (list 1 2 3))
```

```
'#&(1 2 3)
> (box "hello")
'#&"hello"
```

- You can get the contents out of a box with `unbox`:

```
> (unbox (box 5))
5
> (unbox (box (lambda (x) x)))
#<procedure>
> (define box1 (box 5))
> (unbox box1)
5
```

- You can update a box with `set-box!`:

```
> (set-box! box1 111)          ; contained 5...
> (unbox box1)                 ; ...now contains 111
111
```

You can make a box's contents *be itself*:

```
> (set-box! box1 box1)
#0='#&#0#
> box1
#0='#&#0#
```

This "line noise" is Racket trying to "draw" a diagram in which the box points back to itself:

- `#0=` means "I am labelling this box 0";
- `'#&` is Racket's usual "here is a box, whose contents follow";
- `#0#` is a reference back to the box labelled 0.

It may be easier to understand if we make a circular "list" (the term "list" often implies that there are no cycles):

```
> (set-box! box1 (list 1 2 box1))
> box1
#0='#&(1 2 #0#)
```

We can understand this as: "Here is a box labelled 0, and inside the box is a list whose first element is 1, whose second element is 2, and whose third element is the box labelled 0."

```
> (set-box! box1 (list 1 2 3 box1 5 6))
> box1
#0='#&(1 2 3 #0# 5 6)
```

Now the box contains a list, whose 4th element points back to the box itself.

We will use this technique to construct a recursive closure: a closure whose environment $env2$ binds an identifier u to *that same closure*.

11.3.2 Adding a recursive closure

Our *recursive closure* Clo-Rec will differ from the the previous closure Clo: the environment will be in a Racket box.

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
  [Clo (env Env?) (e E?)]                ; not in concrete syntax
  [Clo-Rec (box-env box?) (e E?)]        ; not in concrete syntax
  )
```

Also, because we will use Clo-Rec as a closure around Rec expressions, if we try to evaluate a Clo-Rec we will evaluate its body, rather than treating it as a value (as we did with Clo).

To make this work, we need to change Env-id to evaluate the resulting expression, because the resulting expression might be a Clo-Rec.

I'm not completely sure this is the best or only way to do this—I found it fairly easy to get something that "worked" for good, terminating Fun code, but harder to make expressions that should be nonterminating actually not terminate...

11.3.3 Rules for recursive closures

$\boxed{env \vdash e \Downarrow v}$ Under environment env,
Fun expression e evaluates to value v

$$\frac{}{env \vdash (\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Env-num}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Num}\ n1) \qquad env \vdash e2 \Downarrow (\mathsf{Num}\ n2)}{env \vdash (\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2)}\ \text{Env-add}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Num}\ n1) \qquad env \vdash e2 \Downarrow (\mathsf{Num}\ n2)}{env \vdash (\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2)}\ \text{Env-sub}$$

$$\frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{lookup(env, x) = e \qquad env \vdash e \Downarrow v}{env \vdash (\mathsf{Id}\ x) \Downarrow v}\ \text{Env-id} \qquad \frac{lookup(env, x)\ \text{undefined}}{env \vdash (\mathsf{Id}\ x)\ \text{unknown-id-error}}\ \text{Env-unknown-id}$$

$$\frac{}{env \vdash (\mathsf{Lam}\ x\ e1) \Downarrow \big(\mathsf{Clo}\ env\ (\mathsf{Lam}\ x\ e1)\big)}\ \text{Env-lam} \qquad \frac{}{env \vdash (\mathsf{Clo}\ env_{old}\ e) \Downarrow (\mathsf{Clo}\ env_{old}\ e)}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{old} \vdash eB \Downarrow v}{env \vdash (\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Env-app}$$

$$\frac{env_{old} \vdash e \Downarrow v}{env \vdash (\mathsf{Clo\text{-}rec}\ env_{old}\ e) \Downarrow v}\ \text{Env-clo-rec} \qquad \frac{\overbrace{u{=}(\mathsf{Clo\text{-}rec}\ env2\ e), env}^{env2} \vdash e \Downarrow v}{env \vdash (\mathsf{Rec}\ u\ e) \Downarrow v}\ \text{??Env-rec}$$

$$\frac{\mathcal{E} \Rightarrow \big(u{=}(\mathsf{Clo\text{-}rec}\ \mathcal{E}\ e), env\big) \vdash e \Downarrow v}{env \vdash (\mathsf{Rec}\ u\ e) \Downarrow v}\ \text{Env-rec}$$

12 State

12.1 State

All of our Fun dialects have had only *immutable* bindings: During evaluation, once an identifier is bound to an expression, its meaning cannot change—it will have the same meaning as long as it is in scope (and not shadowed by another binding).

12.1.1 Classifying languages

Many languages have *mutable* state in some form:

- *By default, and idiomatic*: Fortran, Algol-60, Lisp, C, C++, Java, Smalltalk, ...
- *By default, but less idiomatic*: Racket
- *Not by default*: Standard ML, OCaml
- *By simulation*: Haskell

The line between "functional" and "imperative" is fuzzy, but I think most people would draw it somewhere around Racket. The line between "purely functional" and "impurely functional"—*purity* meaning a "lack of side effects (such as state)"—is usually drawn between ML and Haskell. That line is also subject to debate, however.

Starting from the top of the list, in languages like Java, most features are mutable by default (unless `const` is given).

Racket occupies a strange position in this space: fundamental binding operations like **define** and **let** are mutable, but "good Racket style" discourages you from exploiting this. A few language features in Racket, including lists, are genuinely immutable by default (Racket also has mutable lists, but not by default; as we saw in our discussion of classifying languages, different languages often provide the same behaviours and differ only in which behaviour is the default).

The ML languages are fairly consistent in being immutable by default. A value bound by a `let` in SML or OCaml cannot be mutated. Both languages do have features similar to Racket's boxes, but these must be used explicitly; the default is immutability. An exception that puts OCaml slightly nearer the top of the page: strings are mutable, as you can see from the following interaction with OCaml.

```
# let s = "abcd" ;;
val s : string = "abcd"
# String.set s 2 'r' ;;
- : unit = ()
# s ;;
- : string = "abrd"
```

(2016 update: I believe this has been changed in the latest version of OCaml.)

Haskell is usually considered "pure" or "purely functional", though there is debate about this too, partly because some people argue that nontermination is a side effect. In practice, Haskell has ample support (features like the appallingly named "monads") for an imperative style of programming. (As an aside, the techniques used to *implement* Haskell depend on mutable state!)

12.1.2 Defining state

The particular form of state we'll add to Fun is called *references* (using the ML terminology). A *reference* or *ref* is essentially a pointer to a *cell* (or "ref cell") whose contents can be mutated.

Modelling refs in a dynamic semantics requires significant changes. Now that we have an environment-based semantics, the environment env might seem to be a logical place to store the current contents of ref cells. That turns out to be a bad idea—we want the state of ref cells to survive a lexical scope, but not the state of binders (see the question below)—so instead we'll create a new animal, a *store* S. Like an environment, a store is basically a list:

Stores	S	$::=$ $\emptyset$	empty store
		$\mid \ell \triangleright v, S$	location ℓ points to cell with contents v, followed by store S

Let's extend the concrete and abstract syntax.

Expressions	$\langle$E$\rangle$	$::=$	...
		$\mid$	{Ref $\langle$E$\rangle$}
		$\mid$	{Deref $\langle$E$\rangle$}
		$\mid$	{Setref $\langle$E$\rangle$ $\langle$E$\rangle$}

The intended semantics is:

- {Ref ⟨E1⟩} evaluates ⟨E1⟩ and returns the location of a new cell (think of a location as a pointer);
- {Deref ⟨E1⟩} evaluates ⟨E1⟩, which must evaluate to a location, and returns the contents of the cell at that location;
- {Setref ⟨E1⟩ ⟨E2⟩} evaluates ⟨E1⟩, which must evaluate to a location ℓ, then evaluates ⟨E2⟩ and puts that resulting value into the cell at location ℓ.

The abstract syntax is extended correspondingly:

```
(define-type E
  ...
  [Ref (initial-contents E?)]
  [Deref (loc-expr E?)]
  [Setref (loc-expr E?) (new-contents E?)]
```

We aren't quite done, though: a `Ref` is how we *create* a new cell, not a *pointer* to a new cell. So we need one more variant:

```
(define-type E
  ...
  [Ref (initial-contents E?)]
  [Deref (loc-expr E?)]
  [Setref (loc-expr E?) (new-contents E?)]
  [Location (locsym symbol?)]
```

Racket has a built-in function called `gensym` that we'll use when we need a new location, to get a "fresh" symbol.

The point of a store is that its contents are mutable: evaluating an expression may change the store. So we need both an *input* store and an *output* store.

$\boxed{env; S \vdash e \Downarrow v; S'}$ Starting in environment env and store S, evaluating e produces value v and updated store S'

$$\frac{env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{??SEnv-ref}$$

What is ℓ? It really doesn't matter, as long as it isn't already in S1. The judgment "ℓ fresh for S1" means that ℓ is not already mapped by S1.

$$\frac{\ell \text{ fresh for } S1 \qquad env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{SEnv-ref}$$

When a new feature (like Ref) leads us to change the judgment form, we need to check two things:

- We can write rules for the new features.
- We can update the rules for old features.

We seem to have a rule for the new feature Ref, so we should try to update an old rule. We'll do the rule for Pair. We first need to update Eval-pair to use environments. Since pairs don't bind identifiers, this is straightforward:

$$\frac{env \vdash e1 \Downarrow v1 \qquad env \vdash e2 \Downarrow v2}{env \vdash (\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2)}\ \text{Env-pair (stateless version)}$$

$$\frac{env; S \vdash e1 \Downarrow v1; S1 \qquad env; S1 \vdash e2 \Downarrow v2; S2}{env; S \vdash (\mathsf{Pair}\ e1\ e2) \Downarrow (\mathsf{Pair}\ v1\ v2); S2}\ \text{SEnv-pair}$$

This version of Env-pair says that, to evaluate a pair, we first evaluate $e1$ under the given store S, producing a (possibly) changed store S1; then, we evaluate $e2$ under S1, producing another store S2, which is the store produced by the entire evaluation of $(\mathsf{Pair}\ e1\ e2)$.

Following this pattern of passing stores along from premise to premise means that when we draw a derivation tree, and draw a line (really a curve) from the conclusion's starting store (to the left of the turnstile $\vdash$) to the conclusion's result (to the right of the semicolon), the line looks like a thread. The store is "threaded through" the derivation tree.

Unlike previous evaluation rules for Pair, SEnv-pair specifies that an interpreter must evaluate the two expressions $e1$ and $e2$ in a particular order. The expression $e2$ *cannot* be evaluated before $e1$, because we need to evaluate $e1$ to know what S1 is.

(Adding the full, tedious set of error-handling rules to the old big-step semantics, or to the environment-based semantics without state, would also enforce this order. But now, it is enforced within the non-error rule. Small-step semantics also enforces this, using evaluation contexts $\mathcal{C}$.)

Before we try to update all the old rules for this different evaluation judgment, we should figure out how to evaluate the other new features:

$$\frac{\ell \text{ fresh for S1} \qquad env; S \vdash e \Downarrow v; S1}{env; S \vdash (\mathsf{Ref}\ e) \Downarrow (\mathsf{Location}\ \ell); \ell \triangleright v,\ S1}\ \text{SEnv-ref}$$

$$\frac{env; S \vdash e \Downarrow (\mathsf{Location}\ \ell); S2 \qquad \textit{lookup-loc}(S2, \ell) = v}{env; S \vdash (\mathsf{Deref}\ e) \Downarrow v; S2}\ \text{SEnv-deref}$$

$$\frac{env; S \vdash e1 \Downarrow (\mathsf{Location}\ \ell); S1 \qquad env; S1 \vdash e2 \Downarrow v2; S2 \qquad \textit{update-loc}(S2, \ell, v2) = S3}{env; S \vdash (\mathsf{Setref}\ e1\ e2) \Downarrow v2; S3}\ \text{SEnv-setref}$$

The idea of *update-loc* is that $\textit{update-loc}(S2, \ell, v2) = S3$ where S3 is the same as S2, but with $\ell \triangleright \cdots$ replaced by $\ell \triangleright v2$. For example, if

$$S2 \quad = \quad \ell1 \triangleright (\mathsf{Num}\ 1), \ell2 \triangleright (\mathsf{Num}\ 0), \emptyset$$

then

$$\textit{update-loc}(S2, \ell2, (\mathsf{Num}\ 2)) \quad = \quad \ell1 \triangleright (\mathsf{Num}\ 1), \ell2 \triangleright (\mathsf{Num}\ 2), \emptyset$$

Question: Why do we need a separate store? What goes wrong if we use the environment to store ref cells?

Because then we would end up with a similar problem as not doing a freeness check during substitution: we could access an identifier that should be out of scope.

$$ePair \quad = \quad \big(\mathsf{Pair}\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Id}\ x))\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x))\big)$$

Here, the second instance $(\mathsf{Id}\ x)$ is not in the scope of $(\mathsf{Let}\ x \ldots)$. But (if the environment contains ref cells), we have to thread the environment through; otherwise, the second component of the following pair $ePair'$ would be unable to see the effects of the first component. We expect $ePair'$ to evaluate to $(\mathsf{Pair}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1))$ because the first component changes the contents of r from $(\mathsf{Num}\ 0)$ to $(\mathsf{Num}\ 1)$:

$$ePair' \quad = \quad \Big(\mathsf{Let}\ r\ (\mathsf{Ref}\ (\mathsf{Num}\ 0))\ \big(\mathsf{Pair}\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Setref}\ r\ (\mathsf{Num}\ 1)))\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 2)\ (\mathsf{Deref}\ r))\big)\Big)$$

If we thread the environment through, then in $ePair$, the binding of $(\mathsf{Id}\ x)$ would survive and could be used outside its scope; if we don't thread the environment through, $ePair'$ wouldn't behave as expected.

12.1.3 First implementation: `env-state.rkt`

We can **define-type** `Store` following the pattern of `Env`, and update `env-interp` to take *and* return a store. It's quite irritating, because Racket doesn't provide great support for returning pairs of things—I had to **define-type** `Config` to represent both an expression—the value v being returned in $env; S1 \vdash e \Downarrow v; S2$—and the output store S2. But it can be done, and there are no big surprises.

Some of this could be done more easily in ML or Haskell, because those languages have more general "pattern matching" than **type-case**, so adjacent **type-case**s can be combined in one. We still have to look up locations in the store, and update a cell by constructing a new store with different contents for that one cell.

Question: Racket has boxes! Why not just use those, instead of going to all this trouble?

Good question. One answer is that we want to know that our interpreter follows the rules (is "sound with respect to the rules"). The code is annoying to read, but the correspondence to the rules is clear. If we use Racket's boxes as our locations, the code becomes much simpler, but we have to trust that Racket's semantics for boxes matches our rules.

I'm pretty sure it does match the rules, which is why I wrote another version of the interpreter that *does* use Racket boxes (`env-state-direct.rkt`—the word "direct" refers to using Racket's boxes directly).

Another answer is that, while we can (I think!) use Racket's boxes to correctly represent Fun's refs, we are depending on Racket's idea of a store being the same as ours. What if we wanted to allow "time travel" in Fun, where we could "checkpoint" an old store and "rewind" to it later? Racket—as far as I know—doesn't have that feature. So we'd need to either figure out how to checkpoint Racket boxes, or use the `env-state.rkt` representation where we don't use boxes.

This leads us to...

12.1.4 Second implementation: `env-state-direct.rkt`

In this interpreter, the operations on the `Store` **define-type** are simply calls to Racket's `unbox` and `set-box!`. Instead of reflecting the "threading" of the store in our interpreter, we assume that Racket's store behaves in that same way. (Again, I'm pretty sure it does.)

12.2 Translation dictionary

(Section added for lecture, 2016-11-14.)

	Racket	**Fun**	**C**	**C++**	**Java**
allocate	`(box e)`	(Ref *e*)	`malloc`	`new`	`new`
get	`(unbox e)`	(Deref *e*)	`*e`	`*e`	`e.fld`
set	`(set-box! e)`	(Setref *e*1 *e*2)	`*e1 = e2`	`*e1 = e2`	`e1.fld = e2`

The Java column is approximate: a Java object has any number of instance variables, rather than just one.

13 Lazy evaluation

13.1 Evaluation strategies: review and update

13.1.1 Review

Earlier in the course, we came up with two strategies for evaluating function application (App $e1$ $e2$): the *expression strategy*, in which the function argument $e2$ is substituted for x in the body of (Lam x eB), and the *value strategy*, in which the function argument $e2$ is evaluated immediately and the resulting value $v2$ is substituted for x in eB.

Under the expression strategy, we evaluate $e2$ as many times as (Id x) is evaluated; under the value strategy, we evaluate $e2$ exactly once. The value strategy is usually more efficient than the expression strategy, but the expression strategy is more efficient if (Id x) is not evaluated. For example, in

$$\big(\mathsf{App}\ (\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Num}\ 0))\ e2\big)$$

there is no (Id x) in the body of the Lam, so (under the expression strategy) the argument $e2$ will never be evaluated.

Exercise 26. Give a slightly larger (and hopefully less contrived) example of a function application for which the expression strategy is more efficient than the value strategy. (Hint: apply a function of *two* arguments, that is, (Lam x (Lam y ...)).)

13.1.2 Update for environment-based evaluation

We have mostly used the value strategy, and we stayed with that strategy in developing the environment-based evaluation rule for App, Env-app; for clarity, we'll now call that rule Env-app-value:

$$\frac{env \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ \mathrm{x}\ e\mathrm{B}))\qquad env \vdash e2 \Downarrow v2\qquad \mathrm{x}{=}v2, env_{old} \vdash e\mathrm{B} \Downarrow v}{env \vdash (\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Env-app-value}$$

To facilitate experimenting, I'm leaving the value-strategy application App alone, but adding an expression-strategy application App-expr.

To implement the expression strategy, we can use

$$\frac{env \vdash e1 \Downarrow (\mathsf{Clo}\ env_{old}\ (\mathsf{Lam}\ \mathrm{x}\ e\mathrm{B}))\qquad \mathrm{x}{=}(\mathsf{Clo}\ env\ e2), env_{old} \vdash e\mathrm{B} \Downarrow v}{env \vdash (\mathsf{App\text{-}expr}\ e1\ e2) \Downarrow v}\ \text{Env-app-expr}$$

As we did for Lam, we are using a closure to save the current environment so that when (Id x) is evaluated, we can evaluate $e2$ under env rather than some later environment.

For historical reasons, a closure that is used in this way is called a *thunk*.
The rule Env-app-expr (for App-expr) is implemented in `env-lazy.rkt`.
Examples:

```
; for app-expr, do we evaluate the argument twice?
(unparse (interp (parse '{App-expr {Lam x {+ x x}} {+ 1 2}})))

; does it work when the argument is a Lam?
(unparse (interp (parse '{App-expr {Lam f {App f 5}} {Lam y {+ y y}}})))

; does it work when the argument has a free variable (z)?
(unparse (interp (parse '{Let z 100
                              {App-expr {Lam f {App f 5}}
                                        {Lam y {+ y z}}}})))

; are we still doing lexical scope?
(unparse (interp (parse '{Let z 100
                              {App-expr {Lam f {Let z 444 {App f 5}}}
                                        {Lam y {+ y z}}}})))
```

13.2 Lazy evaluation

Environments make it possible to implement a third evaluation strategy, which I think is better than the expression strategy. Whether it's better than the value strategy is unclear.

It's useful to pause and relate my terminology in this course to terminology that you may come across elsewhere. I invented my own terminology because I think it's less confusing, but you should be aware of the more common usage:

invention	CPSC 311 name	"formal" names	"popular" names	other names
1950s	value strategy	call-by-value, CBV	eager evaluation, strict evaluation	[applicative order]
1960	expression strategy	call-by-name, CBN	by name	[normal order]
1971–76	lazy evaluation	call-by-need	lazy evaluation	

Brackets, e.g. "[applicative order]", indicate that there is no consensus that those terms are being used accurately, and that many people will (probably correctly) object and reserve those terms for different concepts. I mention them because you may come across them, and they aren't *entirely* wrong: applicative order is *more like* the value strategy than it is like the expression strategy.

Also, people often say "evaluation order" rather than "evaluation strategy". But I prefer "strategy", at least in 311, because it's not just the *order* in which expressions are evaluated: it's also whether they're evaluated at all.

13.2.1 Overview

From a distance, lazy evaluation looks like the expression strategy:

- function arguments are not evaluated immediately;
- function arguments are only evaluated when used.

The difference from the expression strategy is in what happens when the argument is evaluated, *if* it is evaluated:

- The expression strategy evaluates the argument, but doesn't remember the result. So if it sees (Id x) again, it evaluates the argument again.
- Lazy evaluation remembers the result of evaluating the argument. If it sees (Id x) a second (or third...) time, it returns the result without evaluating it again.

We can't use the environment to remember the result, however, because the environment is only passed *up* the evaluation derivation, not "threaded through". But earlier, we added support for mutable references (boxes), which live in a store that *is* threaded through! So we can use the store to remember our results.

(By the way, this is roughly how lazy evaluation actually works in languages that use it, like Haskell.)

13.2.2 Rules

We want to put the result of evaluating an argument in the store; since the store is what survives leaving a lexical scope, we also need to remember in the store *whether* we have evaluated that argument. The environment will *refer* to the store, using a location.

I left out the store from the above rules, so let's put that in Env-app-expr and then rewrite Env-app-expr to be lazy.

$$\frac{env; S \vdash e1 \Downarrow \big(\text{Clo } env_{old}\ (\text{Lam x } eB)\big); S1 \qquad x{=}(\text{Clo } env\ e2), env_{old}; S1 \vdash eB \Downarrow v; S2}{env; S \vdash (\text{App-expr } e1\ e2) \Downarrow v; S2}\ \text{Env-app-expr}$$

I'll use "Lazy-thk" for the thunk we're creating.

$$\frac{env; S \vdash e1 \Downarrow \big(\text{Clo } env_{old}\ (\text{Lam x } eB)\big); S1 \quad \ell \text{ fresh for } S1 \quad \begin{array}{l} x{=}(\text{Lazy-ptr } \ell), env_{old}; \\ \ell{\triangleright}(\text{Lazy-thk } env\ e2), S1 \vdash eB \Downarrow v; S2 \end{array}}{env; S \vdash (\text{App-lazy } e1\ e2) \Downarrow v; S2}\ \text{Env-app-lazy}$$

In the second premise of Env-app-lazy:

1. We bind x to (Lazy-ptr ℓ). Since the environment isn't mutable, x will *always* be bound to (Lazy-ptr ℓ) for the entire time that x is in scope.
2. We extend the store with a new location ℓ containing (Lazy-thk env $e2$).

Note that we haven't evaluated $e2$ yet—and if evaluating eB does not evaluate (Id x), we never will.

When we evaluate (Id x), we will look it up (Env-id) and evaluate what we find. The environment (not the store!) has x=(Lazy-ptr ℓ), so we find (Lazy-ptr ℓ) and evaluate that.

We need a rule for the case where we haven't evaluated the argument yet. In that case, looking up ℓ in the store will give a Lazy-thk. We evaluate $e2$ under the environment that Env-app-lazy saved, resulting in a value v, and use *update-loc* to replace ℓ with v:

$$\frac{\textit{lookup-loc}(S, \ell) = (\text{Lazy-thk } env_{\text{arg}}\ e2) \qquad env_{\text{arg}}; S \vdash e2 \Downarrow v; S1 \qquad \textit{update-loc}(S1, \ell, v) = S2}{env; S \vdash (\text{Lazy-ptr } \ell) \Downarrow v; S2}\ \text{Env-lazy-ptr}$$

We also need a rule for the case where we already applied Env-lazy-ptr to ℓ. In that case, looking up ℓ in the store will *not* give a Lazy-thk, but some value—the v that we got while applying Env-lazy-ptr.

$$\frac{\textit{lookup-loc}(S, \ell) = v \qquad v \neq (\text{Lazy-thk } \cdots\ \cdots)}{env; S \vdash (\text{Lazy-ptr } \ell) \Downarrow v; S}\ \text{Env-lazy-ptr-done}$$

```
(unparse (interp (parse '{App-lazy {Lam x {+ x x}} {- 10 1}})))
```

13.2.3 Ideology

Evaluation strategy, like typing, is one of the most enduring controversies in programming language design. The controversy began the moment there was more than one evaluation order:

> The first call-by-name language, Algol 60, also supported call-by-value. It seems that call-by-value was the language committee's preferred default, but Peter Naur, the editor of the Algol 60 report, independently reversed that decision—which he said was merely one of a "few matters of detail". A committee member, F.L. Bauer, said this showed that Naur "had absorbed the Holy Ghost after the Paris meeting... there was nothing one could do... it was to be swallowed for the sake of loyalty." (From Dunfield (2015), "Elaborating evaluation-order polymorphism"; quotations from Wexelblat (1981), *History of Programming Languages I*.)

(There was also some argument about whether Naur had independently decided to include recursion in Algol-60, but my reading is that he didn't do that—the committee had agreed to support recursion, but may have had arguments over details.)

Later developments continued to be full of ideology. Lazy evaluation (under the name call-by-need) was introduced in a 1971 PhD thesis, and first implemented (twice, mostly independently, I believe) in 1976. One of the 1976 papers has the rather opinionated title "CONS Should Not Evaluate its Arguments", only softened slightly in the paper itself: "we have uncovered a critical class of elementary functions which probably should never be treated as strict: the functions which allocate or *cons*truct data structures." I suspect that not all ML programmers would agree. Nor would users (at least, designers) of Lisp, the language used in the 1976 paper: neither Lisp nor its descendants Scheme and Racket are lazy.

These controversies are (partly) grounded in legitimate disagreements about design tradeoffs between the value strategy (eager evaluation) and lazy evaluation:

- The value strategy trades speed in one, possibly uncommon case—the case where the argument isn't used—for simplicity.
- Lazy evaluation trades simplicity for speed, but also trades space for speed: the many thunks that have to be created take up space (and slow down garbage collection) even if they are never evaluated. Reasoning about how much space is used is difficult; for example, I believe that the Haskell community relies on *space profilers* to debug "space leaks" that result from thunks being built that are never needed.

- Lazy evaluation sometimes trades actual improvement for apparent improvement: if the expression whose evaluation is being avoided is simple, it would be faster to evaluate it without creating a thunk—*even if the argument is never used*.

There are certainly cases where lazy evaluation is superior, but opponents of lazy evaluation argue that these cases can be handled by explicit programmer-controlled laziness instead. That is, laziness should be an option that must be asked for explicitly, rather than the default.

13.2.4 Function application vs. the whole language

The rules we've developed add lazy function application, without changing any other language constructs. Languages with built-in laziness usually don't stop there. For example, Haskell is entirely lazy: if you add two expressions with +, the addition won't be performed unless the result is "demanded" (such as by printing the value to the user).

14 Subtyping

14.1 Subtyping

In Typed Fun, every expression either has no type (the typing judgment $\Gamma \vdash e : \cdots$ cannot be derived; equivalently, `typeof` returns `#false`) or has a unique type, which is the A such that $\Gamma \vdash e : A$, or equivalently, the type A returned by `typeof`.

Thus, types are *non-overlapping*: if $A \neq B$, then the set of expressions that have type A are disjoint from the expressions that have type B.

In everyday life (and mathematics), we classify things rather more elaborately than Typed Fun: an entity or person may belong to several, overlapping categories. Overlapping categories are beyond our concern today; instead, we'll consider categories that are entirely contained within each other, like the diagrams I showed on Monday.

14.1.1 Our first subtyping system

Mathematicians would generally agree that the number represented by writing 2 is

- a positive integer (an n such that $n \geq 0$);
- an integer;
- a rational number (it can be written as a ratio $\frac{2}{1}$);
- a real number; and
- a complex number (whose imaginary part is 0).

Mathematically, the positive integers are a subset of the integers, which are a subset of the rationals, and so on.

Adding a category such as "even integer" would spoil this arrangement: some numbers are even but not positive, and some numbers are positive but not even. Subtyping *can* capture such relationships, but we'll save those for another time.

The above *inclusion relationships* are fine mathematically, but some of them require caution in a programming language. In particular, the leap from rationals to reals is dangerous: computers represent real numbers as floating-point numbers, which are very strange approximations of real numbers. Converting from a rational to a float is liable to result in a number that is close to the rational, but not close enough. To avoid this problem, we won't attempt to claim that rational numbers (as stored in a computer) are a subset of floating-point numbers.

Fortunately, the first two inclusion relationships (positive integers $\subseteq$ integers, and integers $\subseteq$ rationals) are unproblematic. We currently don't have any of these types, however—only num, which includes everything up to and including complex numbers.

In the past, I've glossed over exactly what counts as a num in Fun by waving my hands in the general direction of Racket's notion of a number. I'll be slightly more rigorous now: I'll restrict Fun to rational numbers, steering clear of the problematic leap from rationals to floats (which are fake "reals"), and then wave my hands at the mathematical notion of a rational number (which I hope is the same as Racket's).

Types	A, B	::=	bool	booleans
			$\mid$ A $\rightarrow$ B	functions from A to B
			$\mid$ A $*$ B	products (pairs)
			$\mid$ pos	integers ≥ 0
			$\mid$ int	integers
			$\mid$ rat	rationals

Why would we want to do this? Well, we might want to know that the absolute value of an integer is always positive. Maybe the result of calling an absolute value function is used as an index to a string (see previous lecture notes), and we want to avoid having to check that the index is non-negative. Also, knowing that the index is an integer, rather than an arbitrary rational number, would eliminate an additional check.

(That leaves the last check: that the index is less than the length of the string. Some type systems will get rid of that check too...)

Or, if you prefer, think of these three types as representing a small class hierarchy in an object-oriented language. Some aspects of OO inheritance are already present in this context, so we can use this simpler setting to build up intuition for how to define classes and inheritance.

(I kind of wanted to jump straight into OO-style subtyping, but instead we'll approach that "sideways". Most OO languages combine what I think of as several different features—records (things that have fields/instance variables/methods), inheritance (subtyping), mutability, self-reference—into one, "objects". But these features don't have to appear together, and I believe they can often be better understood separately.)

Adding a type to the grammar isn't useful unless we can give expressions that type. So let's add three typing rules (the third effectively replaces the rule we used to have for num):

$$\frac{n \in \mathbb{Z} \qquad n \geq 0}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{pos}}\ \text{Type-pos} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{int}}\ \text{Type-int} \qquad \frac{n \in \mathbb{Q}}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{rat}}\ \text{Type-rat}$$

Considering just one binary operator, =, will illustrate several aspects of subtyping. Suppose we have a specialized version of Type-binop, just for =:

$$\frac{\Gamma \vdash e1 : \mathsf{rat} \qquad \Gamma \vdash e2 : \mathsf{rat}}{\Gamma \vdash (\mathsf{Binop}\ (\mathsf{Equalsop})\ e1\ e2) : \mathsf{bool}}\ \text{Type-binop-eq}$$

This is very different from Typed Fun: a single expression, like (Num 5), can have more than one type. (In fact, (Num 5) has three different types.) If our entire program is (Num n) for some n, this isn't a problem. But for more realistic programs, we've painted ourselves into a corner. Consider this silly function:

$$\big(\mathsf{Lam}\ x\ \mathsf{pos}\ (\mathsf{Binop}\ (\mathsf{Equalsop})\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big)$$

Ignore how silly this function is. It may be silly, and applying it will always evaluate to (Btrue), but it's still a function that should typecheck. We won't be able to, however. In its derivation we will assume

x : pos, but the premises of Type-binop-eq require (reasonably enough) that the expressions have type rat.

But in fact, every *closed value* (that is, every expression that (1) has no free variables and (2) is a value) that has type pos also has type rat (and int as well): The only closed values of type pos have the form (Num n) where $n \in \mathbb{Z}$, and $\mathbb{Z} \subseteq \mathbb{Q}$, so $n \in \mathbb{Q}$, so by rule Type-rat, $\emptyset \vdash (\mathsf{Num}\ n) : \mathsf{rat}$.

Our next steps are:

- Design *subtyping rules* that define when one type is a subtype of (included in) another type.
- Update our typing rules to make use of the subtyping rules.

Based on the set inclusions $\{n \in \mathbb{Z} \mid n \geq 0\} \subseteq \mathbb{Z}$ and $\mathbb{Z} \subseteq \mathbb{Q}$, we can write our first subtyping rules:

$$\frac{}{\mathsf{pos} <: \mathsf{int}}\ \text{Sub-pos-int} \qquad \frac{}{\mathsf{int} <: \mathsf{rat}}\ \text{Sub-int-rat}$$

In set theory, we know that the subset relation is reflexive (every set is a subset of itself) and transitive (if $S_1 \subseteq S_2$ and $S_2 \subseteq S_3$, then $S_1 \subseteq S_3$). Any good definition of subtyping should have these same properties. The easiest way to ensure this (at least "on paper") is to add two more rules:

$$\frac{}{A <: A}\ \text{Sub-refl} \qquad \frac{A1 <: A2 \qquad A2 <: A3}{A1 <: A3}\ \text{Sub-trans}$$

For now, the only useful application of Sub-trans is to derive pos <: rat:

$$\frac{\dfrac{}{\mathsf{pos} <: \mathsf{int}}\ \text{Sub-pos-int} \qquad \dfrac{}{\mathsf{int} <: \mathsf{rat}}\ \text{Sub-int-rat}}{\mathsf{pos} <: \mathsf{rat}}\ \text{Sub-trans}$$

These four rules (the general rules Sub-refl and Sub-trans, and the rules specific to our numeric types, Sub-pos-int and Sub-int-rat) constitute a pretty good, or at least non-broken, subtyping system. So we can move on to update our typing rules.

14.1.2 Soundness of subtyping

How do we know that a set of subtyping rules makes sense? For typing rules, we talked about *type safety*: if the typing rules say e has type A, and evaluating e produces a value v, that value v *also* has type A. Otherwise, the static and dynamic semantics don't match.

For subtyping, we can define *subtype soundness*:

Definition 27. Subtype soundness holds if, for all values v and types A, B such that $\emptyset \vdash v : A$ *without* using Type-sub, and $A <: B$, then $\emptyset \vdash v : B$ *without* using Type-sub.

Type-sub is a rule we'll develop below, but since the definition doesn't let you use that rule within itself, it's okay that we haven't developed it yet!

So, for example, a rule

$$\frac{}{\mathsf{rat} <: (\mathsf{rat} \to \mathsf{rat})}\ \text{??Sub-rat-arr}$$

violates subtype soundness, because there exists a v (actually, a whole lot of vs) such that

$$\emptyset \vdash v : \mathsf{rat}$$

but *not*

$$\emptyset \vdash v : (\mathsf{rat} \to \mathsf{rat})$$

In fact, *every* value of type rat is a valid counterexample.

14.1.3 Adding subtyping to the type system

Adding subtyping is easy; adding subtyping that can be easily implemented takes some work.

The easy way is to add a single rule, called the *subsumption rule*:

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{Type-sub}$$

Rule Type-sub says that if we determine that e has type A, and A is a subtype of B, then e has type B.

Using Type-sub, we can type the function that gave us trouble before:

$$\dfrac{\dfrac{\dfrac{\dfrac{(x : \mathsf{pos})(x) = \mathsf{pos}}{x : \mathsf{pos} \vdash (\mathsf{Id}\ x) : \mathsf{pos}}\ \text{Type-id} \qquad \dfrac{\checkmark}{\mathsf{pos} <: \mathsf{rat}}}{x : \mathsf{pos} \vdash (\mathsf{Id}\ x) : \mathsf{rat}}\ \text{Type-sub} \qquad \dfrac{\checkmark}{x : \mathsf{pos} \vdash (\mathsf{Id}\ x) : \mathsf{rat}}}{x : \mathsf{pos} \vdash (\mathsf{Binop}\ (\mathsf{Equalsop})\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)) : \mathsf{bool}}\ \text{Type-binop-eq}}{\emptyset \vdash \big(\mathsf{Lam}\ x\ \mathsf{pos}\ (\mathsf{Binop}\ (\mathsf{Equalsop})\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big) : \mathsf{pos} \to \mathsf{bool}}\ \text{Type-lam}$$

Unfortunately, Type-sub has cheerfully destroyed a useful property of the typing rules: they are no longer *syntax-directed*.

Definition 28. A set of typing rules is *syntax-directed* if, for each syntactic form (variant of the abstract syntax), only one rule has a conclusion that potentially matches that form.

(Warning: the term "syntax-directed" is sometimes used loosely, or with a slightly different meaning—but in all the usages I can recall, our typing rules *were* syntax-directed before we added Type-sub, and they are now *not* syntax-directed.)

We exploited this property to write `typeof`. We also exploited a rather similar property to write `interp`: for each variant of the abstract syntax, either one (usually) or *two* rules have a suitable conclusion (the variants with two rules being Ite and, recently, Tree-case). For the variants with two rules, we could figure out which rule to try by evaluating an expression.

Type-sub has broken this property, because Type-sub's conclusion works for *any* expression! No matter what e is, it is *possible* that we will need to use Type-sub. Even more thrillingly, instead of making a recursive call to `typeof` on a smaller expression, Type-sub has us making a recursive call on *the same expression*! Thus, if we implement our typing rules including Type-sub, we must be careful not to try to derive

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash e : \_\_\_} \qquad \_\_\_ <: \_\_\_}{\Gamma \vdash e : \_\_\_}\ \text{Type-sub} \qquad \_\_\_ <: \_\_\_}{\Gamma \vdash e : \_\_\_}\ \text{Type-sub}$$

Fortunately, we never need to apply Type-sub twice in a row, because subtyping is transitive. So if $e = (\mathsf{Num}\ 1)$ and we derived

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (\mathsf{Num}\ 1) : \mathsf{pos}}\ \text{Type-pos} \qquad \dfrac{}{\mathsf{pos} <: \mathsf{int}}\ \text{Sub-pos-int}}{\Gamma \vdash (\mathsf{Num}\ 1) : \mathsf{int}}\ \text{Type-sub} \qquad \dfrac{}{\mathsf{int} <: \mathsf{rat}}\ \text{Sub-int-rat}}{\Gamma \vdash (\mathsf{Num}\ 1) : \mathsf{rat}}\ \text{Type-sub}$$

we could instead have derived

$$\dfrac{\dfrac{}{\Gamma \vdash (\mathsf{Num}\ 1) : \mathsf{pos}}\ \text{Type-pos} \qquad \overset{\checkmark}{\mathsf{pos} <: \mathsf{rat}}}{\Gamma \vdash (\mathsf{Num}\ 1) : \mathsf{rat}}\ \text{Type-sub}$$

Transitivity took care of that problem, but we still need to know when to try to apply Type-sub. Let's try to figure that out.

What does that mean? Well, some of the rules, like Type-pair, just don't care:

$$\dfrac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) : A1 * A2}\ \text{Type-pair}$$

This rule makes no demands on $A1$ and $A2$. We never need to use Type-sub as the last (bottommost) step of deriving $\Gamma \vdash e1 : A1$ or $\Gamma \vdash e1 : A2$. (Note that we might need Type-sub *somewhere* inside the derivations of these premises, but not as the *last* step.)

Other rules do require something about the types. For example, Type-pair-case requires that the type of the scrutinee e be a product type $A1 * A2$. The rule itself doesn't care what $A1$ and $A2$ are, but e has to be some kind of product and not, say, rat or $\mathsf{pos} \to \mathsf{pos}$.

$$\dfrac{\Gamma \vdash e : (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash e\mathtt{Body} : B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ e\mathtt{Body}) : B}\ \text{Type-pair-case}$$

However, here we *also* don't need to use Type-sub, because (for the moment) we don't have any subtyping for products—except reflexivity: $(A1 * A2) <: (A1 * A2)$—so it won't do us any good. If we *did* have subtyping for product types, we *still* wouldn't want to use it!

Suppose we added some rules so that $(\mathsf{pos} * \mathsf{int}) <: (\mathsf{int} * \mathsf{int})$, and then tried to derive

$$\dfrac{\dfrac{\Gamma \vdash e : (\mathsf{pos} * \mathsf{int})}{\Gamma \vdash e : (\mathsf{int} * \mathsf{int})}\ \text{Type-sub} \qquad x1 : \mathsf{int}, x2 : \mathsf{int}, \Gamma \vdash e\mathtt{Body} : B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ \underbrace{(\mathsf{App}\ (\mathsf{Id}\ pow2)\ (\mathsf{Id}\ x1))}_{e\mathtt{Body}}) : B}\ \text{Type-pair-case}$$

where $\Gamma = (pow2 : \mathsf{pos} \to \mathsf{pos})$.

The idea is that $pow2$ is a Fun function such that $(\mathsf{App}\ (\mathsf{Id}\ pow2)\ (\mathsf{Num}\ k))$ returns the kth power of 2, for integers $k \geq 0$. This function only works for nonnegative integer powers (otherwise it would have to deal with cases like raising 2 to the power 1/3), so its type is $\mathsf{pos} \to \cdots$.

(The result of raising 2 to such a power is always a nonnegative integer, so its result type is also pos. But it's the domain of $pow2$ that matters in this example.)

I haven't chosen an e yet; I can use a pair whose first component is a positive integer, and whose second component is an integer:

$$e = (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4))$$

I'm missing a derivation for $\Gamma \vdash e : (\mathsf{pos} * \mathsf{int})$; I'll leave that as an exercise:

Exercise 29. Complete the derivation tree:

$$\dfrac{}{\Gamma \vdash (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4)) : (\mathsf{pos} * \mathsf{int})}\ \text{- - - - - - - - - - -}$$

(If you used Type-sub to do this, try it again without using Type-sub.)

Returning to the above example, and assuming you did the exercise, the derivation tree we have so far is

$$\dfrac{\dfrac{\overset{\checkmark}{\Gamma \vdash e : (\mathsf{pos} * \mathsf{int})}}{\Gamma \vdash e : (\mathsf{int} * \mathsf{int})}\ \text{Type-sub} \qquad \mathsf{x1} : \mathsf{int}, \mathsf{x2} : \mathsf{int}, \Gamma \vdash \mathit{eBody} : \mathsf{B}}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ \mathsf{x1}\ \mathsf{x2}\ \underbrace{(\mathsf{App}\ (\mathsf{Id}\ \mathsf{pow2})\ (\mathsf{Id}\ \mathsf{x1}))}_{\mathit{eBody}}) : \mathsf{B}}\ \text{Type-pair-case}$$

Now we want to derive

$$\mathsf{x1} : \mathsf{int}, \mathsf{x2} : \mathsf{int}, \Gamma \vdash \underbrace{(\mathsf{App}\ (\mathsf{Id}\ \mathsf{pow2})\ (\mathsf{Id}\ \mathsf{x1}))}_{\mathit{eBody}} : \mathsf{B}$$

Trying Type-app, we get

$$\dfrac{\mathsf{x1} : \mathsf{int}, \mathsf{x2} : \mathsf{int}, \Gamma \vdash (\mathsf{Id}\ \mathsf{pow2}) : (\mathsf{pos} \to \mathsf{pos}) \qquad \mathsf{x1} : \mathsf{int}, \mathsf{x2} : \mathsf{int}, \Gamma \vdash (\mathsf{Id}\ \mathsf{x1}) : \mathsf{pos}}{\mathsf{x1} : \mathsf{int}, \mathsf{x2} : \mathsf{int}, \Gamma \vdash \underbrace{(\mathsf{App}\ (\mathsf{Id}\ \mathsf{pow2})\ (\mathsf{Id}\ \mathsf{x1}))}_{\mathit{eBody}} : \mathsf{pos}}\ \text{Type-app}$$

The first premise can be derived with Type-id, recalling that $\Gamma = (\mathsf{pow2} : \mathsf{pos} \to \mathsf{pos})$.

But the second premise can't be derived! We know that x1 is an integer, but we don't know that it's a positive integer. We knew that the scrutinee had type $\mathsf{pos} * \mathsf{int}$, but we forgot that information when we used Type-sub.

The lesson here is not to use Type-sub unless you really need to. One place where we do need to use Type-sub is Type-binop-eq, which requires that the expressions being compared have type rat.

14.2 Developing subtyping

Once we decide that the values of a type can also be values of another, larger type, subtyping becomes another (nested) step in the recipe of adding a feature to the language:

1. Extend the concrete syntax.
2. Extend the abstract syntax.
3. Extend the dynamic semantics (e.g. evaluation rules).
4. For a typed language, extend the static semantics (e.g. typing rules):
   (a) For a language with subtyping, extend the subtyping rules.
5. (Not in CPSC 311.) Prove desirable properties of the language (e.g. type safety).

(Steps 3–4 need not be done in that order.)

We're adding subtyping rather late in the game, but we can go back through the features we've built up, adding subtyping to them.

14.2.1 Product types (pair types)

For products, the subtyping rule works "pairwise":

$$\frac{A1 <: B1 \qquad A2 <: B2}{(A1 * A2) <: (B1 * B2)}\ \text{Sub-product}$$

For example, every pair of an int and a bool is also a pair of a rat and a bool:

$$\frac{\dfrac{}{\mathsf{int} <: \mathsf{rat}}\ \text{Sub-int-rat} \qquad \dfrac{}{\mathsf{bool} <: \mathsf{bool}}\ \text{Sub-refl}}{(\underbrace{\mathsf{int}}_{A1} * \underbrace{\mathsf{bool}}_{A2}) <: (\underbrace{\mathsf{rat}}_{B1} * \underbrace{\mathsf{bool}}_{B2})}\ \text{Sub-product}$$

14.2.2 Lists

For lists, we can follow the pattern of pairs (for this purpose, the Lispish notion that a list is "really" a pair is not wrong):

$$\frac{A <: B}{(\mathsf{list}\ A) <: (\mathsf{list}\ B)}\ \text{Sub-list}$$

For example, every list of positive integers is also a list of integers.

14.2.3 Trees

For trees, we can also follow the pattern of pairs.

$$\frac{A <: B}{(\mathsf{tree}\ A) <: (\mathsf{tree}\ B)}\ \text{Sub-tree}$$

For example, every tree whose keys are positive integers is also a tree whose keys are integers.

Notice that for products, lists and trees, the subtyping in the premise(s) "goes the same way" as the subtyping in the conclusion: In Sub-tree, A appears on the left of $<:$ in the conclusion, and in the premise. In Sub-product, $A1$ appears on the left of $<:$ in the conclusion, and also on the left of $<:$ in a premise; $A2$ works similarly.

Because the subtyping goes the same way, Sub-product, Sub-list and Sub-tree are said to be *covariant*.

14.2.4 Functions

A function type $A1 \to A2$ has two types inside it, a domain of inputs $A1$ and a range of outputs (or "codomain") $A2$. Following the pattern of Sub-product, we get

$$\frac{A1 <: B1 \qquad A2 <: B2}{(A1 \to A2) <: (B1 \to B2)}\ \text{??Sub-arr}$$

However, to quote John Reynolds, "As usual, something funny happens at the left of the arrow." (This is one of the enduring truths of programming languages.) Using ??Sub-arr, we can derive

$$\frac{\dfrac{}{\mathsf{int} <: \mathsf{rat}}\ \text{Sub-int-rat} \qquad \dfrac{}{\mathsf{bool} <: \mathsf{bool}}\ \text{Sub-refl}}{(\mathsf{int} \to \mathsf{bool}) <: (\mathsf{rat} \to \mathsf{bool})}\ \text{??Sub-arr}$$

This should mean that, if we expect to be given a function of type (rat → bool), we should be happy with a function of type (int → bool). But a function of type (int → bool) is only half as good as one of type (rat → bool), because a function whose domain is rat can be applied to any rational number, while a function whose domain is int can only be applied to integers.

Informally, ??Sub-arr is validating false advertising: a function that only handles integers should not be able to pass itself off as a function that handles all rational numbers.

More formally, rule ??Sub-arr violates the Liskov[–Wing] (1994) "Subtype Requirement" (often called the "Liskov substitution principle"):

> Let $\varphi(x)$ be a property provable about objects x of type T.
> Then $\varphi(y)$ should be true for objects y of type S where S is a subtype of T.

As our property Φ we can essentially use type safety: a property of functions f of type rat → bool is that, when applied to any value of type rat, certain errors will not occur.

However, this property is *not* true of all functions g of type int → bool: type safety tells us that, for any function g : (int → bool), if we apply g to any value of type int, certain errors will not occur.

But that doesn't tell us that those errors will not occur *for arguments that are not integers*. Here, it's useful to recall something from our treatment of strings. We added an expression Nth that returned the nth character in a string:

$$\frac{eS \Downarrow (\mathsf{Str}\ s1) \qquad eIdx \Downarrow (\mathsf{num}\ n) \qquad n \in \mathbb{Z} \qquad n \geq 0 \qquad n < \mathit{len}(s1)}{(\mathsf{Nth}\ eS\ eIdx) \Downarrow (\mathsf{Str}\ s1_n)}\ \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \cancel{\mathsf{num}}\ \mathsf{rat}}{\Gamma \vdash (\mathsf{Nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

Since we only had a generic num type, whenever we evaluated Nth we had to check that the index evaluated to a number that was (1) an integer, (2) ≥ 0, and (3) less than the length of the string.

Now that we have a type int, we can remove the check for n being an integer from Eval-nth, provided Type-nth checks that eIdx is an int and not merely a rat:

$$\frac{eS \Downarrow (\mathsf{Str}\ s1) \qquad eIdx \Downarrow (\mathsf{num}\ n) \qquad \cancel{n \in \mathbb{Z}} \qquad n \geq 0 \qquad n < \mathit{len}(s1)}{(\mathsf{Nth}\ eS\ eIdx) \Downarrow (\mathsf{Str}\ s1_n)}\ \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{string} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \cancel{\mathsf{num}}\ \cancel{\mathsf{rat}}\ \mathsf{int}}{\Gamma \vdash (\mathsf{Nth}\ eS\ eIdx) : \mathsf{string}}\ \text{Type-nth}$$

Let g be the function

$$(\mathsf{Lam}\ x\ \mathsf{int}\ (\mathsf{Nth}\ (\mathsf{Str}\ \texttt{"hello"})\ (\mathsf{Id}\ x)))$$

which has type int → bool. Applying g to a rat, say 2.5, will lead to an error that should be impossible: taking the 2.5th character of a string.

So rule ??Sub-arr doesn't work. (Some people call the relation described by ??Sub-arr "naïve subtyping". I do not approve: subtyping that uses ??Sub-arr is not a form of subtyping that is naïve, it's *not subtyping at all!* If you want even more disapproval of this term, consult Ron Garcia.)

A rule that does work is this one, which is *contravariant* in the domain, meaning the subtyping "goes the other way" in the premise for the function domains A1 and B1:

$$\frac{B1 <: A1 \qquad A2 <: B2}{(A1 \to A2) <: (B1 \to B2)}\ \text{Sub-arr}$$

14.2.5 Refs

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

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{Type-sub} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash (\mathsf{Id}\ x) : A}\ \text{Type-var}$$

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

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

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

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

$$\frac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) : A1 * A2}\ \text{Type-pair} \qquad \frac{\Gamma \vdash e : A1 * A2 \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eBody) : B}\ \text{Type-pair-case}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Let}\ x\ e\ eBody) : B}\ \text{Type-with} \qquad \frac{u : B, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{rec}\ u\ B\ e) : B}\ \text{Type-rec}$$

$$\frac{}{\Gamma \vdash (\mathsf{list\text{-}empty}\ A) : \mathsf{list}\ A}\ \text{Type-empty} \qquad \frac{\Gamma \vdash e1 : A \qquad \Gamma \vdash e2 : \mathsf{list}\ A}{\Gamma \vdash (\mathsf{list\text{-}cons}\ e1\ e2) : \mathsf{list}\ A}\ \text{Type-cons}$$

$$\frac{\Gamma \vdash e : \mathsf{list}\ A \qquad \Gamma \vdash eEmpty : B \qquad xh : A, xt : \mathsf{list}\ A, \Gamma \vdash eCons : B}{\Gamma \vdash (\mathsf{List\text{-}case}\ e\ eEmpty\ xh\ xt\ eCons) : B}\ \text{Type-list-case}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{Ref}\ e) : \mathsf{ref}\ A}\ \text{Type-ref} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ A}{\Gamma \vdash (\mathsf{Deref}\ e) : A}\ \text{Type-deref} \qquad \frac{\Gamma \vdash e1 : \mathsf{ref}\ A \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{Setref}\ e1\ e2) : A}\ \text{Type-setref}$$

Figure 14.1 Typing rules for Typed Fun with pairs, lists, and refs

14.2.5.1 Subtyping for refs

Following the pattern of list, we might write a covariant rule for references:

$$\frac{A <: B}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{??Sub-ref}$$

By this rule, (ref int) <: (ref rat). However, if you expect something of type ref rat and I give you an expression of type (ref int), you can use setref to replace the reference's contents with 3.5 (because, to you, it is a ref rat and you can assign any rat to it).

So we might try contravariance:

$$\frac{B <: A}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{??Sub-ref-2}$$

Now, however, if you expect something of type (ref int) and deref it, expecting an int, you may be disappointed: By ??Sub-ref-2, (ref rat) <: (ref int). But the contents of (ref rat) could be 3.5 or any rational number, not necessarily an integer.

The covariant rule ??Sub-ref works fine with deref, but not with setref; the contravariant rule ??Sub-ref-2 works fine with setref, but not with deref. So the covariant rule enforces a necessary condition for deref, and the contravariant rule enforces a necessary condition for setref. Therefore, a correct rule is:

$$\frac{A <: B \qquad B <: A}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{Sub-ref}$$

which enforces *both* conditions.

(We might try to "optimize" this rule by replacing the premises with $A = B$. This is probably okay for this system, but doesn't work for all type systems, so I'd rather leave it as is.)

The following may be a useful additional explanation, particularly if you understand contravariant subtyping for function types $A1 \to A2$. We can think of a reference as an object with two methods, called deref and setref:

- The deref "method" has no arguments (we are thinking of this, for the moment, as a class method, so the reference to "self" or "this" is implicit), and returns (for a reference of type (ref A)) a value of type A.

  So we can think of the type of deref as $() \to A$, where $()$ represents taking zero arguments.

- The setref "method" takes one argument, of type A (assuming the reference has type (ref A)). It also returns the value of the argument. So we can think of the type of setref as $A \to A$.

Thus, the deref "method" has type $() \to A$ and setref has type $A \to A$. According to the contravariant rule for functions, Sub-arr, we can compare the types of the deref method of a reference of type (ref A) and the deref method of a reference of type (ref B) as follows:

$$\frac{() <: () \qquad A <: B}{(() \to A) <: (() \to B)}\ \text{Sub-arr}$$

The second premise here matches the covariant premise of Sub-ref. (Regardless of whatever () is, exactly, the first premise is derivable using Sub-refl.)

For setref, we get

$$\frac{\mathsf{B} <: \mathsf{A} \qquad \mathsf{A} <: \mathsf{B}}{(\mathsf{A} \to \mathsf{A}) <: (\mathsf{B} \to \mathsf{B})}\ \text{Sub-arr}$$

The second premise here is something of an accident: we happened to decide that setref should return the new contents just written to the reference. If we said, instead, that setref returned "nothing", which we seem to be writing as $()$, then we would have

$$\frac{\mathsf{B} <: \mathsf{A} \qquad () <: ()}{(\mathsf{A} \to ()) <: (\mathsf{B} \to ())}\ \text{Sub-arr}$$

14.2.6 Upper bounds

Something I hadn't thought of by Monday's lecture: there are a few more places where we need to use Type-sub. We need to use it in Type-ite; otherwise, `typeof` will return `false` for the expression

$$(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ -1))$$

This is because $(\mathsf{Num}\ 1)$ has type pos, and $(\mathsf{Num}\ -1)$ has type int, but $\mathsf{pos} \neq \mathsf{int}$. So when we implement Type-ite, we need to find the *upper bound* of the types of the eThen and eElse branches:

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

$$\frac{\Gamma \vdash e : B \qquad B = \mathsf{bool} \qquad \Gamma \vdash \mathit{eThen} : A1 \qquad \Gamma \vdash \mathit{eElse} : A2 \qquad A1 = A2}{\Gamma \vdash (\mathsf{Ite}\ e\ \mathit{eThen}\ \mathit{eElse}) : A1}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B \qquad B <: \mathsf{bool} \qquad \Gamma \vdash \mathit{eThen} : A1 \qquad A1 <: A \qquad \Gamma \vdash \mathit{eElse} : A2 \qquad A2 <: A}{\Gamma \vdash (\mathsf{Ite}\ e\ \mathit{eThen}\ \mathit{eElse}) : A}\ \text{Type-ite*}$$

This last version of Type-ite, marked *, is really just the original Type-ite with three uses of Type-sub:

$$\frac{\dfrac{\Gamma \vdash e : B \qquad B <: \mathsf{bool}}{\Gamma \vdash e : \mathsf{bool}}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash \mathit{eThen} : A1 \qquad A1 <: A}{\Gamma \vdash \mathit{eThen} : A}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash \mathit{eElse} : A2 \qquad A2 <: A}{\Gamma \vdash \mathit{eElse} : A}\ \text{Type-sub}}{\Gamma \vdash (\mathsf{Ite}\ e\ \mathit{eThen}\ \mathit{eElse}) : A}\ \text{Type-ite}$$

That is, Type-ite* is an easier rule to implement, but Type-ite* isn't adding any power to the type system. (It's harder, actually, to prove that Type-ite* isn't *taking anything away* from the type system. But I'm pretty sure it isn't.)

We also need to do this in some other rules, such as Type-list-case, so I wrote a function `upper-bound` that takes two types A and B, and returns A if B <: A, and B if A <: B. See the updated version of `subtyping.rkt`.

15 Records

Updated 2016–11–20 with an explanation of `upper-bound` and another exercise.

15.1 Records

The first part of these notes is handwritten:
`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/notes/scan-2016-11-18.pdf`

15.1.1 Record syntax

(See the first scanned page.)

Note the syntactic sugar `{Record {x y pos}}` for several fields with the same type.

The expression (Dot e y) evaluates e to a record, and returns the field named y.

Field names like x and y are not bindings; they don't have a scope. The only way to use a field name is in (Dot e y). If (Id y) appears in the e in (Dot e y), it must be bound in the usual way by a Let, Lam, Pair-case, etc.

We won't define substitution for this system, but if we did, the field names would never be affected by substitution. We *would* need to substitute within the *contents* of the fields.

A record with two fields is roughly the same as a pair, if we provided only Fst and Snd for accessing the parts of the pair, instead of Pair-case. (Something like Pair-case on a record would be reasonable; in Pascal, a similar feature was called `with`. Pattern matching in languages like ML works on records, too.)

Question: Can we have two fields with the same name?

Yes, in different record types. But you shouldn't make a record with two fields with the same name. My implementation doesn't check for this, and it's probably not too hard, but I don't want to commit to saying it's easy when I haven't done it.

Records can be nested inside other records, or placed into refs, or used as arguments or results to functions. We're designing records as an *orthogonal feature*: nothing about the record type, or record expressions, forces us to have any other particular feature in the language. We could have a language with records but not functions, or with records but not refs, and so on.

15.1.2 Width subtyping

(See the second scanned page.)

All we can do with a record is access a field using (Dot e y). It shouldn't matter if other fields are present; they can't affect the value of the field y.

Thus, if we define a function (top of the page) that expects, as its argument, a record with one field $x : \mathsf{pos}$, it should be okay to pass a record with additional fields.

To do that, we need to use subtyping, so we can show that

$$(\mathsf{record}\ \ x{:}\mathsf{pos}, y{:}\mathsf{pos})\ \mathsf{<:}\ (\mathsf{record}\ \ x{:}\mathsf{pos})$$

The effect is kind of like subclassing in Java, at least, the part of subclassing that is about adding instance variables to the subclass that aren't present in the superclass.

This also (maybe) justifies the rather strange type $(\mathsf{record}\ )$, which is the type of (record), the record with no fields: it's a little like Java's `Object`.

15.1.3 Depth subtyping

Another form of subtyping that's useful for records is "depth subtyping", which says that a record with one field y, of type A, is a subtype of a record with one field y of type B, provided that A is a subtype of B. This is reminiscent of subtyping for pairs (the Sub-product rule).

15.1.3.1 Upper bounds

For example, depth subtyping allows us to pass a record of type $(\mathsf{record}\ \ y{:}\mathsf{pos})$ to a function that expects $(\mathsf{record}\ \ y{:}\mathsf{rat})$. According to depth subtyping, this is allowed because $\mathsf{pos}\ \mathsf{<:}\ \mathsf{rat}$.

In an earlier version of `typeof` with subtyping, several branches called a function `upper-bound`. For example, the branch for Ite called `upper-bound` on the types of `eThen` and `eElse`. This is necessary because we might need to use Type-sub. For example, our typing rules show that $(\mathsf{Num}\ 3)$ has type pos and -5 has type int:

$$\frac{3 \in \mathbb{Z} \qquad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\ \text{Type-pos} \qquad \frac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ -5) : \mathsf{int}}\ \text{Type-int}$$

But if we try to use the above derivations as the second and third premises of Type-ite, we get stuck:

$$\frac{\emptyset \vdash (\mathsf{Bfalse}) \qquad \dfrac{3 \in \mathbb{Z} \qquad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\ \text{Type-pos} \qquad \dfrac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ -5) : \mathsf{int}}\ \text{Type-int}}{\emptyset \vdash (\mathsf{Ite}\ (\mathsf{Bfalse})\ \underbrace{(\mathsf{Num}\ 3)}_{\mathit{eThen}}\ \underbrace{(\mathsf{Num}\ -5)}_{\mathit{eElse}}) : ???}\ \text{Type-ite}$$

Type-ite requires the *same* type in each branch. On paper (given enough time to think about it), we can fix this by using Type-sub to "forget" that—in addition to being an integer—$(\mathsf{Num}\ 3)$ is a positive integer. That gives us the same type, int, for both *eThen* and *eElse*, which allows us to apply Type-ite.

$$\frac{\emptyset \vdash (\mathsf{Bfalse}) \qquad \dfrac{\dfrac{3 \in \mathbb{Z} \qquad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\ \text{Type-pos} \qquad \dfrac{}{\mathsf{pos}\ \mathsf{<:}\ \mathsf{int}}\ \text{Sub-pos-int}}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{int}}\ \text{Type-sub} \qquad \dfrac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ -5) : \mathsf{int}}\ \text{Type-int}}{\emptyset \vdash (\mathsf{Ite}\ (\mathsf{Bfalse})\ \underbrace{(\mathsf{Num}\ 3)}_{\mathit{eThen}}\ \underbrace{(\mathsf{Num}\ -5)}_{\mathit{eElse}}) : \mathsf{int}}\ \text{Type-ite}$$

In the code for `typeof`, we don't have the luxury of thinking about where to use Type-sub. Instead, we always use the *upper bound* of the types of *eThen* and *eElse*. In effect, `typeof` is implementing a rule that looks like this:

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash \mathit{eThen} : \mathit{AThen} \qquad \Gamma \vdash \mathit{eElse} : \mathit{AElse}}{\Gamma \vdash (\mathsf{Ite}\ e\ \mathit{eThen}\ \mathit{eElse}) : \mathit{upper\text{-}bound}(\mathit{AThen}, \mathit{AElse})}\ \text{Type-ite-upperbound}$$

The idea of *upper-bound*(A1, A2) is that it returns a type that is a supertype of both A1 and A2, that is, if *upper-bound*(A1, A2) = B then A1 <: B and A2 <: B.

We actually want it to be the *least upper bound*: for example, rat is *an* upper bound of pos and int, because pos <: rat and int <: rat, but it is not the *least* upper bound because int is also a supertype of pos and int.

The earlier version of `upper-bound` just checked whether one of the types (A1, A2) was a subtype of the other. It worked for pos and rat, because pos is a subtype of int, and int is a subtype of rat. for rat and int, because int is a subtype of rat.

15.1.3.2 Upper bounds of record types

With records, the subtyping relationship is more complicated: (record x:pos, y:pos) is a subtype of (record x:pos). Also, (record x:pos, z:pos) is a subtype of (record x:pos). But (record x:pos, y:pos) is neither a subtype of (record x:pos, z:pos), nor a supertype of it.

To compute the upper bound of

(record x:pos, y:pos)

and

(record x:pos, z:pos)

we need to take all the fields in common, that is, $\{x, y\} \cap \{x, z\} = \{x\}$; for each of those fields, make a recursive call to find the upper bound of the types. In this example, there is one field in common, x, and it has the same type pos, so we take the upper bound of pos and pos, which is pos.

If we compute the upper bound of

(record x:int, y:pos)

and

(record x:rat, z:pos)

we get (record x:rat), because the upper bound of int and rat is rat.

Exercise 30. Suppose we decided to add a language feature

(Setfield *e* x *e2*)

that **updates** the field x in record *e* with *e2*. Setting aside the question of how to define evaluation for Setfield, would width subtyping and depth subtyping still make sense? If not, what kind of subtyping for records would we need instead?

Exercise 31. Does the implementation of `upper-bound` in `a5.rkt` really do what we want? Try to find an example of types A1 and A2 such that

```
(upper-bound A1 A2)
```

returns `#false` even though, according to the subtyping rules on a5, there is a type B such that A1 <: B and A2 <: B.

If you find an example that involves record types, try to find another example that does not.

15.2 Downcasts

A Downcast is an odd expression; certainly, its typing rule (Type-downcast) is odd. It says that if e has some type B, then (Downcast A e) has type A. It checks that A is a subtype of B. But that's the *opposite* of Type-sub!

$$\frac{A <: B \qquad \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Downcast}\ A\ e) : A}\ \text{Type-downcast} \qquad \frac{env; S1 \vdash e \Downarrow v; S2 \qquad \emptyset \vdash v : A}{env; S1 \vdash (\mathsf{Downcast}\ A\ e) \Downarrow v; S2}\ \text{SEnv-downcast}$$

When (Downcast A e) is evaluated, we check, *during evaluation*, that the value v resulting from the expression e inside (Downcast A e) actually does have type A. If v doesn't have type A, then no evaluation rule applies, and the interpreter raises an error.

This is motivated by the following example. Suppose we had strings, with an expression (Idx $e\mathsf{Str}$ $e\mathsf{Idx}$) that indexes into $e\mathsf{Str}$. The index $e\mathsf{Idx}$ must evaluate to (Num n), and n must be (1) an integer, (2) positive ($n \geq 0$), and (3) less than the length of the string that $e\mathsf{Str}$ evaluates to. Since we have a type specifically for positive integers, pos, conditions (1) and (2) can be enforced in the typing rule for idx via a premise $\Gamma \vdash e\mathsf{Idx} : \mathsf{pos}$.

$$\frac{\Gamma \vdash e\mathsf{Str} : \mathsf{string} \qquad \Gamma \vdash e\mathsf{Idx} : \mathsf{pos}}{\Gamma \vdash (\mathsf{Idx}\ e\mathsf{Str}\ e\mathsf{Idx}) : \mathsf{string}}\ \text{Type-idx}$$

But suppose we have, in our Fun program, an identifer x of type int, and we want to use x to index into a string. We can use Ite to check whether x is positive (the "else" branch in the expression shown), so checks (1) and (2) in the interpreter will always succeed.

$$\mathsf{x} : \mathsf{int}, \mathsf{s} : \mathsf{string} \vdash \Big(\mathsf{Ite}\ \big(\mathsf{Binop}\ <\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Num}\ 0)\big)\ (\mathsf{Str}\ \texttt{"bad"})\ \big(\mathsf{Downcast}\ \mathsf{pos}\ (\mathsf{Idx}\ (\mathsf{Id}\ \mathsf{s})\ (\mathsf{Id}\ \mathsf{x}))\big)\Big) : \mathsf{string}$$

Unfortunately, the typing rule with premise $\Gamma \vdash e\mathsf{Idx} : \mathsf{pos}$ doesn't let us use (Id x) as that index, because all we know is that x has type int, not that x has type pos.

We can use Downcast to make this work:

$$\big(\mathsf{Idx}\ (\mathsf{Id}\ \mathsf{s})\ (\mathsf{Downcast}\ \mathsf{pos}\ (\mathsf{Id}\ \mathsf{x}))\big)$$

The downcast check $\emptyset \vdash v : \mathsf{pos}$ will always succeed, because $\big(\mathsf{Binop}\ <\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Num}\ 0)\big)$ must have evaluated to (Bfalse).

16 Type inference

16.1 Type inference

When we started doing typing, we encountered the rule

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ e) : A \to B}\ \text{?Type-lam}$$

which we couldn't implement, because to make the recursive call to `typeof`, we needed to extend the typing context Γ (`tc`) with $x : A$, but we didn't know what A was.

As a workaround, we added the type A to the syntax of Lam:

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ e) : A \to B}\ \text{?Type-lam}$$

Now we'll show how to do (a simple form of) *type inference*, which *infers* the type A without making the programmer write it.

The general idea is to use A as a placeholder, and update it once we know what A needs to be. On paper, this is not too difficult: we write A and B instead of actual types, and leave empty boxes off to the side of the derivation. (See the scanned page.) Initially, these boxes are blank because we don't yet know what A and B are, but from Type-add, we can figure out that since $(\mathsf{Id}\ x)$ has type A, and Type-add needs A needs to be num, then we should use num. At this point, we write num in the box labelled A.

From the conclusion of Type-add, we also see that B is num.

Even though we wrote $A \to B$ in the conclusion, we know that $A = \mathsf{num}$ and $B = \mathsf{num}$, so we have really derived

$$\emptyset \vdash (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))) : \mathsf{num} \to \mathsf{num}$$

In fact, everywhere we wrote A in the derivation, we should now interpret A as num. By writing num in the box for A, we have updated or "mutated" A. This suggests a way to implement this technique: represent A as a Racket box that is either "blank" or "filled in" with a type. To "fill in" the box, we can use Racket's `set-box!`.

Extending the **define-type** for `Type` (see type-inference.rkt), we have

```
(define-type Type
    [t/num]                                ; Num
    [t/bool]                               ; Bool
    [t/-> (domain Type?) (range Type?)]    ; {-> domain range}
    [t/var (var box?)]
)
```

We will represent a blank box on paper by a box containing `#false`, and a filled-in box by a box containing a `Type`.

(I tried to use a Racket "box contract" `box/c` to specify this, but ran into trouble with "impersonators" and "chaperones". Yes, really.)

16.1.1 Equating types

The `type=?` function provides a starting point for a central mechanism of type inference, *unification*. Our extended version of `type=?` is called `type=?!`.

But unlike `type=?`, which is only asking "are these types equal?", the new function `type=?!` is asking "can these types *be made* equal?"

If a mathematician accosts you and asks, "Is x plus 1 equal to 5?" the correct answer is "I don't know". But if she asks you to solve the equation

$$x + 1 = 5$$

you should answer "$x = 4$". More generally, given an equation, you can try to solve all the variables in it.

If the equation has no variables, then you are just doing arithmetic. So, for types without any `t/vars`, this new function `type=?!` will work just like `type=?`: if the types are literally the same, it will return `#true`, otherwise `#false`.

The difference is in how `type=?!` works on variables `t/var`:

- If the first type is a variable whose box (call it α) contains `#false` (meaning "blank" or "unknown"), then we are trying to solve the equation

  $$\alpha = B$$

  where we don't have a solution for α. But the solution is right there: let $\alpha = B$. So in this case, we use `set-box!` to replace the `#false` inside the box α with B.

  Boxes are mutable and global, so this operation effectively "rewrites" any other occurrences of `t/var` α in the derivation.

- If the second type is a variable whose box (call it β) contains `#false`, then we have a situation symmetric to the one above: we are trying to solve

  $$A = \beta$$

  So we use `set-box!` to put A inside β.

- If the first type is a variable that has been solved, its box α contains a type $A0$. That is, we know that $\alpha = A0$, and we want to try to make $\alpha = B$, so we try to make $A0 = B$.

- Similarly, if the second type is variable that has been solved, its box β contains a type $B0$. We know that $\beta = B0$, and we want to make $A = \beta$, so we try to make $A = B0$.

Since `type=?!` will be our mechanism for solving type variables, we need to update `typeof` (which is now called `infer`) to use `type=?!` more often. For example, the old `infer` sometimes used `type-case` with a `t/num` branch to check whether a type `A1` was a `num`. Now the type might be a `t/var`, so instead, `infer` calls `(type=?! (t/num) A1)`.

16.1.1.1 The "occurs check"

As given, this technique works most of the time, but it will not work for this Fun expression:

$$(\mathsf{Lam}\ x\ (\mathsf{App}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)))$$

In the `lam` branch, we create a `t/var` for x, and recursively call `infer` to infer the type of the body $(\mathsf{App}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))$.

In the `app` branch of `infer`, we create a type `(t/-> A1 B)` where A1 and B are `t/vars`. Here, A1 will be made equal to A (the type of x). But we need the type of $(\mathsf{Id}\ x)$ to be equal to $A1 \to B$, that is, equal to `(t/-> A1 B)`. The type of $(\mathsf{Id}\ x)$ is A, so (since $A = A1$) we are trying to make this equation hold:

$$A \to B \ = \ A$$

where A is unsolved. So (the original version of) `type=?!` sets the contents of A's box to $A \to B$. This creates a cyclic "type" that makes something (I'm not sure what, and lack the patience to figure it out) loop forever.

But it should never be possible for $A \to B$ to equal A. For any such types without `t/vars`, `type=?` would have returned `#false`.

The solution is something called an "occurs check", which checks whether the `t/vars` occurs in the other type. For our example, that amounts to checking whether A occurs in $A \to B$. It does occur, so we return `#false`.

Adding the occurs check led to another problem, which is that A occurs in A. So I added another check, done before the occurs check, to see whether both A and B are the same type variable; if they are, `type=?!` returns `#true`. (A mathematician accosts you and asks if x equals x. You should say "yes". You should especially say "yes" if the mathematician is also an Objectivist, because "*A* is *A*".)

2015-11-25

TYPE INFERENCE:

(lam x ~~A~~ e)
(rec u ~~B~~ e)

$$\frac{x{:}A, \Gamma \vdash e : B}{\Gamma \vdash (\text{lam } x\ e) : A \to B}\ \text{Type-lam [?]}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash (\text{id } x) : A}\ \text{Type-id} \qquad \frac{\Gamma \vdash e1 : \text{Num} \quad \Gamma \vdash e2 : \text{Num}}{\Gamma \vdash (\text{add } e1\ e2) : \text{Num}}\ \text{Type-add} \qquad \frac{\Gamma \vdash e1 : A1 \quad (A1 = \text{Num}) \quad \Gamma \vdash e2 : A2 \quad (A2 = \text{Num})}{\Gamma \vdash (\text{add } e1\ e2) : \text{Num}}$$

A = Num
B = Num

$$\frac{\dfrac{(x{:}A) \in x{:}A, \emptyset}{x{:}A, \emptyset \vdash (\text{id } x) : A} \qquad \dfrac{(x{:}A) \in x{:}A, \emptyset}{x{:}A, \emptyset \vdash (\text{id } x) : A}}{x{:}A, \emptyset \vdash (\text{add } (\text{id } x)\ (\text{id } x)) : \text{Num}}$$

$$\emptyset \vdash (\text{lam } x\ (\text{add } (\text{id } x)\ (\text{id } x))) : A \to B$$

Num → Num

Types:

+ (t/var b)

where b is a box that contains either #false (meaning "unknown", like A=[ ]), or a type.

(The type might also be a t/var.)

For the above example:

$$\emptyset \vdash (\text{lam } x\ (\text{add } (\text{id } x)\ (\text{id } x))) : A \to B$$

(t/-> (t/var •) (t/var •))

A [(t/num)] B [t/num]

17 Bidirectional typing

Remark. Parts of these notes were adapted from my McGill lecture notes. A few "ML-isms" remain, such as a distinction between "expressions" and "declarations"; these are syntactically distinct in SML, and were also syntactically distinct in the language that was developed in the McGill course (a tiny version of SML). I chose to keep this distinction, since mainstream languages often make a similar distinction; for example, in C and Java, local variable declarations are distinct from statements and expressions.

In [Typed] Fun, the closest thing to a declaration is a Let expression; in versions that have Let*, each binding in a Let* could be considered a declaration.

Also, these notes consistently use "variable" rather than "identifier".

Finally, the explanation of the "subsumption rule" is a little out of place, because at McGill, I introduced bidirectional typing *before* subtyping.

17.1 Introduction

When we started doing typing, we encountered the rule

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ e) : A \to B}\ \text{?Type-lam}$$

which we couldn't implement, because to make the recursive call to `typeof`, we needed to extend the typing context Γ (`tc`) with $x : A$, but we didn't know what A was.

As a workaround, we added the type A to the syntax of Lam:

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ e) : A \to B}\ \text{?Type-lam}$$

Here, the A in $(\mathsf{Lam}\ x\ A\ e)$ is a *type annotation*. Many typed languages do not force you to write A in this situation. One technique for doing that is *type inference*, used in ML, OCaml, Haskell, and other languages (even C++, which now has an `auto` keyword). In these languages, you still need to write types in some places, such as module interfaces and some uses of references.

Not having to write type annotations has some drawbacks. Programmers are deprived of a form of high-grade documentation ("high-grade" because it is formal and machine-checked, unlike English comments which are vague when not outright wrong). There's also the problem that more advanced, precise type systems—those that can statically check array accesses, data structure invariants, etc., etc.—*require* (at least some) annotations, as type inference is undecidable! Last but not least, without type annotations, there is no record of the programmer's intent except the declarations themselves, and so type error messages often fail to highlight the genuine source of the error.

At the other extreme, we could require a type annotation on every variable declaration (as is required in many "mainstream" languages). This is quite tedious, since the type must be written even when it's obvious.

Bidirectional typing lies between the extremes of type inference and mainstream type checking. Type annotations are required for *some* expressions, and therefore on some declarations, particularly function declarations where the documentation aspect of type annotations is especially important. Unlike type inference, which works fine for relatively simple type systems but then "flames out", bidirectional typing is a good foundation for powerful, precise type systems that can check more program properties (such as, again, array accesses). It seems only a matter of time before it is widely used in practice, though as with so much of academic programming languages research, the time involved may well be measured in decades.

17.2 Two directions of information

The main idea: Instead of persisting in trying to figure out the type of an expression on its own as `typeof` does, we alternate between figuring out or *synthesizing* types and *checking* expressions against types we already know.

In terms of judgments, bidirectionality replaces the judgment

$$\Gamma \vdash e : A \qquad \text{"under assumptions in the context } \Gamma\text{, the expression } e \text{ has type } A\text{"}$$

with two different judgments:

$$\Gamma \vdash e \Rightarrow A \quad \text{read "under assumptions in } \Gamma\text{, the expression } e \text{ synthesizes type } A\text{"}$$

$$\Gamma \vdash e \Leftarrow A \quad \text{read "under assumptions in } \Gamma\text{, the expression } e \text{ checks against type } A\text{"}$$

The difference between these judgments is in which parts of the judgment are *inputs* and which are *outputs*. When we want to derive $\Gamma \vdash e \Rightarrow A$, we only know Γ and e: the point is to figure out the type A *from* e, kind of like we did in `typeof`. But when deriving $\Gamma \vdash e \Leftarrow A$, we already know A, and just need to make sure that e does conform to (check against) the type A.

17.3 Typing rules

Two ideas will help us design the rules for deriving bidirectional typing judgments:

(1) We can't use information we don't have.

(2) We should use information we do have.

The second observation leads to our first typing rule, for variables. First, we should define (as a BNF grammar) the form of Γ, which represents contexts (sometimes called, confusingly, environments) of typing assumptions.

$$\begin{array}{rcll} \Gamma & ::= & \emptyset & \text{Empty context} \\ & | & x : A, \Gamma & \text{Context } \Gamma \text{ plus the assumption that variable } x \text{ is of type } A \end{array}$$

And now, the rule for typing variables. It says that if we have assumed x to have type A, because $x : A$ is given in Γ, then x synthesizes type A.

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow A}\ \text{Synth-var}$$

17.3.1 Functions

If we Apply a function, we need the type of the function. But when we create a function by writing (Lam x e), we don't (yet) know that type! So the rule for applications e1 e2 needs to synthesize the function type *from* the function e1.

On the other hand, in the rule for (Lam x e) we don't yet know what the domain or range of the function should be, so (following observation (1)) we check (Lam x e) against a type that is (somehow) already known.

$$\frac{\Gamma \vdash e1 \Rightarrow A \to B \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) \Rightarrow B}\ \text{Synth-app} \qquad \frac{x : A, \Gamma \vdash e \Leftarrow B}{\Gamma \vdash (\mathsf{Lam}\ x\ e) \Leftarrow (A \to B)}\ \text{Check-lam}$$

In most situations, these rules work well: When applying a function, if the function being applied is just a variable, we can synthesize its type (rule Synth-var) so we can indeed synthesize the type of the function e1 in Synth-app.

$$\frac{\dfrac{(\ldots)(g) = \mathsf{bool} \to \mathsf{num}}{g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{Id}\ g) \Rightarrow \mathsf{bool} \to \mathsf{num}}\ \text{Synth-var} \qquad g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{Bfalse}) \Leftarrow \mathsf{bool}}{g : \mathsf{bool} \to \mathsf{num}, \emptyset \vdash (\mathsf{App}\ (\mathsf{Id}\ g)\ (\mathsf{Bfalse})) \Rightarrow \mathsf{num}}\ \text{Synth-app}$$

Or, if the function being applied is itself a function application, as in

$$\big(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Id}\ f))\ (\mathsf{Id}\ x)\big)$$

(where *twice*, which applies its first argument to its second argument twice, has type (num → num) → num → num) that *also* synthesizes its type (rule Synth-app, applied to twice f), so again we can successfully apply Synth-app. We can also successfully type

$$\big(\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))))\ (\mathsf{Id}\ x)\big)$$

because in Synth-app, we check the argument e2 = (Lam y (Add (Id y) (Id y))) against the domain A = (num → num), which is the type that Check-lam checks the lam against. Here is the derivation, where

$$\Gamma = \mathit{twice} : \underbrace{((\mathsf{num} \to \mathsf{num}) \to \mathsf{num} \to \mathsf{num})}_{A\mathit{twice}}, x : \mathsf{num}$$

$$\dfrac{\dfrac{}{\Gamma \vdash \mathit{twice} \Rightarrow A\mathit{twice}}\ \text{Synth-var} \quad \dfrac{\dfrac{\vdots}{y : \mathsf{num}, \Gamma \vdash (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)) \Rightarrow \mathsf{num} \quad \mathsf{num}{=}\mathsf{num}}\ \text{Check-sub}}{\Gamma \vdash (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))) \Leftarrow (\mathsf{num} \to \mathsf{num})}\ \text{Check-lam}}{\Gamma \vdash (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))) \Rightarrow (\mathsf{num} \to \mathsf{num})}\ \text{Synth-app} \quad \dfrac{\dfrac{}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow \mathsf{num}}\ \text{Synth-var} \quad \mathsf{num}{=}\mathsf{num}}{\Gamma \vdash (\mathsf{Id}\ x) \Leftarrow \mathsf{num}}\ \text{Check-sub}$$

$$\Gamma \vdash (\mathsf{App}\ (\mathsf{App}\ (\mathsf{Id}\ \mathit{twice})\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))))\ (\mathsf{Id}\ x)) \Rightarrow \mathsf{num}\ \text{Synth-app}$$

These rules don't let us immediately apply a lam; for example,

$$\big(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ (\mathsf{Num}\ 5)\big)$$

won't typecheck because Synth-app demands that the lam synthesize, and our only rule for lam, namely Check-lam, doesn't synthesize:

$$\frac{\emptyset \vdash (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))) \not\Rightarrow \qquad \ldots}{\emptyset \vdash \big(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ (\mathsf{Num}\ 5)\big) \not\Rightarrow}\ \text{Synth-app}$$

This restriction is inconvenient for the small examples we often use in 311, but it's not very inconvenient in practice: real code seldom applies a Lam immediately in this way.

17.3.2 "Subsumption"

We actually used an undeclared rule in the example above. When we check $(\mathsf{Id}\ \mathsf{f})$ against $\mathsf{num} \to \mathsf{num}$, we need to derive the judgment $(\mathsf{Id}\ \mathsf{f}) \Leftarrow \mathsf{num} \to \mathsf{num}$. But our only rule for variables is Synth-var, which (assuming $\mathsf{f} : \mathsf{num} \to \mathsf{num}$ is in Γ) derives $\Gamma \vdash (\mathsf{Id}\ \mathsf{f}) \Rightarrow \mathsf{num} \to \mathsf{num}$.

We need a rule that lets us show that an expression checks against a type, provided the expression synthesizes the same type. For reasons related to subtyping, this rule is called *subsumption* and we write it with an explicit comparison between A (the type checked against), and A' (the synthesized type).

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub}$$

17.3.3 Recursive expressions and typing annotations

$$\frac{u : B, \Gamma \vdash e \Leftarrow B}{\Gamma \vdash (\mathsf{Rec}\ u\ e) \Leftarrow B}\ \text{Check-rec} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Anno}\ e\ A) \Rightarrow A}\ \text{Synth-anno}$$

Annotations allow us to turn expressions that don't synthesize a type into expressions that do, by writing the type ourselves. Recall the expression $(\mathsf{App}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ (\mathsf{Num}\ 5))$, which doesn't synthesize a type because $(\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))$ doesn't synthesize. Using an annotation, we can readily synthesize the expression's type:

$$\dfrac{\dfrac{\dfrac{y : \mathsf{num} \vdash (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)) \Leftarrow \mathsf{num}}{\emptyset \vdash (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y))) \Leftarrow \mathsf{num}\to\mathsf{num}}\ \text{Check-lam}}{\emptyset \vdash \left(\mathsf{Anno}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ \mathsf{num}\to\mathsf{num}\right) \Rightarrow \mathsf{num}\to\mathsf{num}}\ \text{Synth-anno} \qquad \dfrac{\dfrac{}{\emptyset \vdash (\mathsf{Num}\ 5) \Rightarrow \mathsf{num}}\ \text{Synth-num} \qquad \mathsf{num} = \mathsf{num}}{\emptyset \vdash (\mathsf{Num}\ 5) \Leftarrow \mathsf{num}}\ \text{Check-sub}}{\emptyset \vdash \left(\mathsf{App}\ \left(\mathsf{Anno}\ (\mathsf{Lam}\ y\ (\mathsf{Add}\ (\mathsf{Id}\ y)\ (\mathsf{Id}\ y)))\ \mathsf{num}\to\mathsf{num}\right)\ (\mathsf{Num}\ 5)\right) \Rightarrow \mathsf{num}}\ \text{Synth-app}$$

17.3.4 Primitive operations

The typing rules for Add and Sub work similarly to the rules for Synth-app, if we think of the operator being applied as a function whose type is known, and the two expressions $e1$ and $e2$ as its arguments.

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

17.3.5 Booleans

$$\frac{}{\Gamma \vdash (\mathsf{Btrue}) \Rightarrow \mathsf{bool}}\ \text{Synth-btrue} \qquad \frac{}{\Gamma \vdash (\mathsf{Bfalse}) \Rightarrow \mathsf{bool}}\ \text{Synth-bfalse}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{bool} \qquad \Gamma \vdash e1 \Leftarrow B \qquad \Gamma \vdash e2 \Leftarrow B}{\Gamma \vdash (\mathsf{Ite}\ e\ e1\ e2) \Leftarrow B}\ \text{Check-ite}$$

17.3.6 Pairs

$$\frac{\Gamma \vdash e1 \Leftarrow A1 \qquad \Gamma \vdash e2 \Leftarrow A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) \Leftarrow (A1 * A2)}\ \text{Check-pair}$$

$$\frac{\Gamma \vdash e \Rightarrow (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash e\mathtt{Body} \Leftarrow B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ e\mathtt{Body}) \Leftarrow B}\ \text{Check-pair-case}$$

17.3.7 Let

In the expression

$$\Big(\mathsf{Let}\ x\ \big(\mathsf{App}\ (\mathsf{Id}\ \mathtt{fact})\ (\mathsf{Num}\ 5)\big)\ \big(\mathsf{Pair}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)\big)\Big)$$

we should be able to figure out (assuming our context Γ contains the typing $\mathtt{fact} : \mathsf{num} \to \mathsf{num}$) that $(\mathsf{Id}\ x)$ has type num, and therefore $\big(\mathsf{Pair}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)\big)$ checks against $\mathsf{num} * \mathsf{num}$.

$$\frac{\Gamma \vdash e1 \Rightarrow A \qquad x : A, \Gamma \vdash e2 \Leftarrow B}{\Gamma \vdash (\mathsf{Let}\ x\ e1\ e2) \Leftarrow B}\ \text{Check-let}$$

17.3.8 Adding more convenience

The rules above can be criticized for requiring too many annotations. For example, even if the body of a Let does synthesize a type, Check-let refuses to utilize that fact, and demands that the type of the body be given already. The same criticism applies to Check-ite, and even Check-pair: the rules above can derive

$$\Gamma \vdash (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 5)) \Leftarrow \mathsf{num} * \mathsf{num}$$

but not

$$\Gamma \vdash (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 5)) \Rightarrow \mathsf{num} * \mathsf{num}$$

This is less of a problem in practice than it might appear: many Lets *can* be checked, such as a Let that is the body of a lam; many pairs are passed as arguments to functions, where their types will be checked.

For the other cases, we can deal with many of these problems fairly easily, by adding Synth- versions of some of the Check- rules.

$$\frac{\Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) \Rightarrow (A1 * A2)}\ \text{Synth-pair} \qquad \frac{\Gamma \vdash e \Rightarrow (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash \mathit{eBody} \Rightarrow B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ \mathit{eBody}) \Rightarrow B}\ \text{Synth-pair-case}$$

$$\frac{\Gamma \vdash e1 \Rightarrow A \qquad x : A, \Gamma \vdash e2 \Rightarrow B}{\Gamma \vdash (\mathsf{Let}\ x\ e1\ e2) \Rightarrow B}\ \text{Synth-let}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{bool} \qquad \Gamma \vdash e1 \Rightarrow A \qquad \Gamma \vdash e2 \Rightarrow A}{\Gamma \vdash (\mathsf{Ite}\ e\ e1\ e2) \Rightarrow A}\ \text{Synth-ite}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{bool} \qquad \Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2 \qquad A1 = A2}{\Gamma \vdash (\mathsf{Ite}\ e\ e1\ e2) \Rightarrow A1}\ \text{Synth-ite}$$

The last two versions of Synth-ite are equivalent.

17.4 Scaling up

New typed languages, and new versions of typed languages, tend to accumulate more and fancier type systems. Type inference works well for very small functional languages. But extending type inference to support a more powerful type system often constitutes a research project in itself!

In particular, type inference has a lot of trouble with subtyping. (Some of the hostility of functional programmers to object-oriented languages may be "sour grapes": typed object-oriented languages have subtyping, while typed functional languages that use type inference don't, partly *because* they use type inference.)

Bidirectional typing easily supports subtyping. In fact, all we have to do is change the Check-sub rule (whose name didn't make sense, because it didn't do any subtyping!) to use <: instead of =:

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub}$$

This avoids the problem of possibly trying to apply Check-sub repeatedly on the same expression: Check-sub's conclusion has $\Leftarrow$, but its premise has $\Rightarrow$, and there is no Synth- rule that uses subtyping.

It also avoids other difficulties we've seen with Type-sub. For example, in bidirectional typing, there is no need for an `upper-bound` function.

Bidirectional typing also easily supports operator overloading, record subtyping, intersection types, and refinement types.

17.5 Typing implemented by `bidir-1.rkt`

Types: num, bool, $A1 * A2$, $A1 \to A2$

$$\frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) \Rightarrow A}\ \text{Synth-var} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \text{Check-sub} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Anno}\ e\ A) \Rightarrow A}\ \text{Synth-anno}$$

$$\frac{\Gamma \vdash e1 \Rightarrow A \to B \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) \Rightarrow B}\ \text{Synth-app} \qquad \frac{x : A1, \Gamma \vdash e \Leftarrow A2}{\Gamma \vdash (\mathsf{Lam}\ x\ e) \Leftarrow A1 \to A2}\ \text{Check-lam}$$

$$\frac{u : A, \Gamma \vdash e \Leftarrow A}{\Gamma \vdash (\mathsf{Rec}\ u\ e) \Leftarrow A}\ \text{Check-rec} \qquad \frac{\Gamma \vdash e1 \Rightarrow A1 \qquad x : A1, \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{Let}\ x\ e1\ e2) \Leftarrow A}\ \text{Check-let}$$

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

$$\frac{}{\Gamma \vdash (\mathsf{Btrue}) \Rightarrow \mathsf{bool}}\ \text{Synth-btrue} \qquad \frac{}{\Gamma \vdash (\mathsf{Bfalse}) \Rightarrow \mathsf{bool}}\ \text{Synth-bfalse}$$

$$\frac{\Gamma \vdash e \Leftarrow \mathsf{bool} \qquad \Gamma \vdash e1 \Leftarrow A \qquad \Gamma \vdash e2 \Leftarrow A}{\Gamma \vdash (\mathsf{Ite}\ e\ e1\ e2) \Leftarrow A}\ \text{Check-ite}$$

$$\frac{\Gamma \vdash e1 \Leftarrow A1 \qquad \Gamma \vdash e2 \Leftarrow A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) \Leftarrow (A1 * A2)}\ \text{Check-pair}$$

$$\frac{\Gamma \vdash e \Rightarrow (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody \Leftarrow A}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eBody) \Leftarrow A}\ \text{Check-pair-case}$$

$$\frac{\Gamma \vdash e1 \Rightarrow A1 \qquad \Gamma \vdash e2 \Rightarrow A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) \Rightarrow (A1 * A2)}\ \text{Synth-pair}$$