CMSC 22100/32100: Programming Languages

Lecture Notes

M. Blume **December 5, 2008**

1 Introduction

What are *programming languages*?

Languages: formal languages, *i.e.*, *sets of strings*

Programming: languages with *semantics*

1.1 Languages as sets

Example: difference terms: Three equivalent ways of defining the syntax of the language:

Inductive definition: L is the smallest set such that:

- $\texttt{0} \in L$
- $\texttt{1} \in L$
- if $e \in L$ and $e' \in L$, then $e{-}e' \in L$

Notice: juxtaposition of symbols (taken from the set $\{\texttt{0}, \texttt{1}, -\}$) is *string concatenation*, which is a monoid operation, meaning that among other things it is *associative*.

BNF: definition using BNF (Backus-Naur-Form), *a.k.a. production rules*, *a.k.a.* "context-free grammar":

$$\begin{array}{lcl} e & \rightarrow & \texttt{0} \\ e & \rightarrow & \texttt{1} \\ e & \rightarrow & e{-}e \end{array}$$

Here: e is a "non-terminal" symbol; `0`, `1`, and $-$ are "terminal" symbols. One of the non-terminals is the designated "start" symbol (so in our case it must be e since that's all we've got). The language is *generated* by taking the start symbols and repeatedly applying one of the production rules, replacing its non-terminal on the left with the corresponding right-hand side. The language is defined to be the set of strings of terminal symbols (non non-terminals!) that can be generated in this fashion.

Frequently used alternative notation, which tends to be more compact:

$$e \quad ::= \quad \mathtt{0} \mid \mathtt{1} \mid e{-}e \tag{1}$$

Inference rules: We use a *judgment* (a *term* encoding a relation, in this case a set) to characterize the expressions of the language. We want to be able to *derive* the judgment $\boxed{e\ \mathbf{exp}}$ if and only if $s \in L$ where e encodes the string s. For the time being we will use strings directly, *i.e.*, we make e identical to s:

$$\frac{}{\mathtt{0}\ \mathbf{exp}}\ \text{ZERO} \qquad \frac{}{\mathtt{1}\ \mathbf{exp}}\ \text{ONE} \qquad \frac{e\ \mathbf{exp} \qquad e'\ \mathbf{exp}}{e{-}e'\ \mathbf{exp}}\ \text{MINUS}$$

Each inference rule has a name, written to its right, so we can refer to it later on. It consists of a horizontal line, separating the *premises* above from the *conclusion* below.

Each premise and each conclusion is a *term template*, *i.e.*, a term possibly containing meta-variables such as e and e' above. A rule containing meta-variables is a finite representation of the (usually infinite) family of *ground* rules (rules without meta-variables) that can be obtained by substituting actual (ground) terms for all meta-variables.

A (ground) judgment A can be *derived* if there exists a set of rules where A is the conclusion of at least one rule, and where every premise in any of the rules is also the conclusion of some (other) rule. We can arrange this set of rules into a tree called the *derivation tree* or *proof tree*.

Example: We can derive the judgment $\mathtt{1} - \mathtt{0} - \mathtt{1}\ \mathbf{exp}$ as follows:

$$\dfrac{\dfrac{}{\mathtt{1}\ \mathbf{exp}}\ \text{ONE} \qquad \dfrac{\dfrac{}{\mathtt{0}\ \mathbf{exp}}\ \text{ZERO} \qquad \dfrac{}{\mathtt{1}\ \mathbf{exp}}\ \text{ONE}}{\mathtt{0} - \mathtt{1}\ \mathbf{exp}}\ \text{MINUS}}{\mathtt{1} - \mathtt{0} - \mathtt{1}\ \mathbf{exp}}\ \text{MINUS}$$

Notice that, as we will see shortly, there is another possible derivation for the same judgment.

1.2 Semantics

So far our language is just a bunch of strings with no meaning. Of course, intuitively we want to think of the $-$ symbol as denoting some sort of arithmetic operation, namely subtraction.

We can try to characterize the meaning of a term by defining a binary relation $\Downarrow$ in such a way that $e \Downarrow n$ can be read as "e evaluates to n." Here e is a term in our language of expressions and n is an integer ($\in \mathbb{Z}$).

$$\frac{}{\mathtt{0} \Downarrow 0}\ \textsc{evzero} \qquad \frac{}{\mathtt{1} \Downarrow 1}\ \textsc{evone} \qquad \frac{e \Downarrow n \qquad e' \Downarrow n'}{e\mathtt{-}e' \Downarrow n - n'}\ \textsc{evminus}$$

Notice that in the conclusion of rule EVMINUS the $-$ between n and n' is the *semantic* minus—subtraction of values in $\mathbb{Z}$ (meaning that if n and n' are numbers, then so is $n - n'$), while the $-$ between e and e' is a syntactic symbol (meaning that if e and e' are terms, then $e\mathtt{-}e'$ is a longer term consisting of the concatenation of e, $-$, and e').

The definition of $\Downarrow$ mirrors the definition of **exp**—and it suffers from the same deficiency, namely that derivations for $e \Downarrow n$ are not uniquely determined by e. This can easily be seen by again considering $\mathtt{1 - 0 - 1}$:

$$\dfrac{\dfrac{}{\mathtt{1} \Downarrow 1}\ \textsc{evone} \qquad \dfrac{\dfrac{}{\mathtt{0} \Downarrow 0}\ \textsc{evzero} \qquad \dfrac{}{\mathtt{1} \Downarrow 1}\ \textsc{evone}}{\mathtt{0 - 1} \Downarrow -1}\ \textsc{evminus}}{\mathtt{1 - 0 - 1} \Downarrow 2}\ \textsc{evminus}$$

vs.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{1} \Downarrow 1}\ \textsc{evone} \qquad \dfrac{}{\mathtt{0} \Downarrow 0}\ \textsc{evzero}}{\mathtt{1 - 0} \Downarrow 1}\ \textsc{evminus} \qquad \dfrac{}{\mathtt{1} \Downarrow 1}\ \textsc{evone}}{\mathtt{1 - 0 - 1} \Downarrow 0}\ \textsc{evminus}$$

The problem is that the ambiguity in the choice of derivations leads to an ambiguity in the answer. In other words, the relation $\Downarrow$ that we defined is not *single-valued*, *i.e.*, it does not represent a *function*.

1.3 Abstract syntax

The root of the problem with ambiguity is that there exist multiple derivations for e **exp**. Each derivation corresponds to a different *parse tree* for the string e. Part of the trouble is that our underlying domain of expressions is a monoid—a mathematical structure where the operation of combining elements into new elements is associative. This operation—which here is string concatenation—does not retain information about the order in which an expression was put together. If we use parentheses informally to indicate that order, then $\mathtt{a(bc)}$ is the same as $\mathtt{(ab)c}$; string concatenation "forgets" the grouping.

But for our semantics the grouping is essential! We want $\mathtt{(1 - 0)-1}$ be treated differently than $\mathtt{1-(0 - 1)}$. Ambiguous parse trees are an important practical problem because programs in actual programming languages are in

fact strings. There exist a variety of methods for disambiguating grammars. However, in this course we will ignore this problem altogether by not considering the strings themselves. Instead, we will consider sets of *terms*, where terms have a hierarchical structure that captures nesting. Roughly speaking, the terms representing programs are the parse trees that we would get when parsing the underlying string representation.

Representing programs as terms is called *abstract syntax*.

1.4 Terms

A *ground term* $c(t_1, \ldots, t_n)$ consists of a *constructor* c of some arity $n \geq 0$ and n other ground terms $t_1, \ldots, t_n$. The constructors are drawn from some finite set. The outermost constructor (here: c) is called the *head constructor* of the overall term. One can visualize ground terms as general trees whose nodes are labeled by constructors.

The qualifier "ground" is used to distinguish ground terms form general terms (*a.k.a.* "term templates") which may also contain meta-variables (see below).

Example: Ground terms representing difference expressions: We have two nullary constructors **zero** and **one** as well as one binary constructor **minus**. The expression $(\texttt{1} - \texttt{0}) - \texttt{1}$ corresponds to the ground term $\mathbf{minus}(\mathbf{minus}(\mathbf{one}, \mathbf{zero}), \mathbf{one})$. Notice that the other possible grouping, namely $\texttt{1} - (\texttt{0} - \texttt{1})$ corresponds to a different term, namely $\mathbf{minus}(\mathbf{one}, \mathbf{minus}(\mathbf{zero}, \mathbf{one}))$.

At the outset, ground terms are *uninterpreted*. Any meaning has to be defined separately on a case-by-case basis.

Notational conventions: To improve readability, we will often deviate from the strict "constructor first, arguments (in parentheses) next" notational scheme for terms. Here are some typical examples:

constants: Terms that consist of nullary (0-argument) constructors and an empty argument list are sometimes called *constants*. For constants we will omit the parentheses and simply write c instead of $c()$.

infix: We often employ infix notation such as $t_1 + t_2$ instead of the arguably more cumbersome $+(t_1, t_2)$. An example of this was the notation $e \Downarrow n$, which is perhaps easier to read than $\Downarrow (e, n)$ or even $\mathbf{evaluates}(e, n)$.

post-fix: In case of unary constructors c (constructors whose arity is 1) we sometimes use post-fix notation $t\ c$ instead of $c(t)$. An example for this was e **exp**, which stands for $\mathbf{exp}(e)$.

mix-fix: For certain constructors that take many arguments it is convenient to employ a "mix-fix" notation. In essence, the constructor name is split into pieces which are then spread out and put between individual arguments. For example, the constructor representing so-called "typing judgments" for a language such as *System F* is usually written as

$$\Delta; \Gamma \vdash e : \tau$$

even though this is just a single constructor (which might be called **typing**) with four arguments (Δ, Γ, e, and τ) so that the above notation is syntactic sugar for $\mathbf{typing}(\Delta, \Gamma, e, \tau)$.[1] In effect, the constructor **typing** is split up into three pieces: ";", "$\vdash$", and ":".

juxtaposition: In the extreme case the constructor is not written at all. This technique is often used for terms that represent function application. Here we write $e_1\ e_2$ to mean $\mathbf{app}(e_1, e_2)$.

General terms: Instead of talking about one ground term at a time, we often want to refer to a whole class of such ground terms that share a common pattern. A *term* is either:

- a *meta-variable* x, or
- has the from $c(t_1, \ldots, t_n)$ where c is an n-ary term constructor and $t_1, \ldots, t_n$ are other terms.

Notice that ground terms are a special case of terms. When we want to emphasize that a term is not necessarily ground, we call it either a *general term* or a *term template*.

1.5 Recursion and Induction on Terms

We can define functions on terms recursively (*a.k.a.* inductively) via sets of equations. For example, a simple evaluation function for difference expressions can be given as:

$$\begin{aligned} \mathrm{val}(\mathbf{zero}) &= 0 \\ \mathrm{val}(\mathbf{one}) &= 1 \\ \mathrm{val}(\mathbf{minus}(e, e')) &= \mathrm{val}(e) - \mathrm{val}(e') \end{aligned}$$

Similarly, we can define functions for determining "size" and "depth" of a difference expression:

$$\begin{aligned} \mathrm{size}(\mathbf{zero}) &= 1 \\ \mathrm{size}(\mathbf{one}) &= 1 \\ \mathrm{size}(\mathbf{minus}(e, e')) &= 1 + \mathrm{size}(e) + \mathrm{size}(e') \end{aligned}$$

$$\mathrm{depth}(\mathbf{zero}) = 0$$

[1] In some situations we get judgments that have even more arguments. For example, judgments describing certain program transformations might look like this:

$$\Delta; \Gamma \vdash e : \tau \rightsquigarrow \hat{e} : \hat{\tau}$$

$$\begin{aligned} \text{depth}(\mathbf{one}) &= 0 \\ \text{depth}(\mathbf{minus}(e, e')) &= 1 + \max(\text{depth}(e), \text{depth}(e')) \end{aligned}$$

Notice that these three functions are defined only for ground terms. Strictly speaking we have to verify that each set of equations makes sense, *i.e.*, that it actually defines a function. In the case of val we have to show that if we can derive equations $\text{val}(t) = n$ and $\text{val}(t) = n'$, then it must be the case that $n = n'$.

1.6 Meta-Substitutions

Substitution is an important topic that we will re-visit frequently. Usually it arises in the context of talking about the variables within a particular programming language that we are defining. Here, however, we are concerned with our meta-language, the language of terms and term templates which we use to talk *about* other languages. The substitution of terms for meta-variables is a meta-substitution. Fortunately, since the language of terms does not have any mechanisms for binding local (meta-)variables, the notion of meta-substitution is quite simple.

A (meta-)substitution is a finite mapping from a set of meta-variables to terms. The set of meta-variables where a substitution σ is defined is called the *domain* of σ and denoted by $\text{dom}(\sigma)$. A *ground substitution* is a substitution whose image consists entirely of ground terms.

Any (meta-)substitution σ (which is defined on meta-variables) can be lifted to a function $\hat{\sigma}$ that works on arbitrary term templates by postulating:

$$\begin{aligned} \hat{\sigma}(x) &= \sigma(x) \\ \hat{\sigma}(c(t_1, \ldots, t_n)) &= c(\hat{\sigma}(t_1), \ldots, \hat{\sigma}(t_n)) \end{aligned}$$

That is, when applying $\hat{\sigma}$ to a term that is not a variable, we leave the constructor unchanged and "push" the substitution into every sub-term. In the case of variables we simply apply the original σ.

To avoid notational clutter we will from now identify $\hat{\sigma}$ with σ and write $\sigma(t)$, implicitly referring to the above definition of $\hat{\sigma}(t)$.

1.7 Rules

A rule is a pair (p, c), where p is a list of *premises* and c is a single *conclusion*.[2] The conclusion c as well as each premise in p is a general term. We could represent the list p itself as a term using two constructors: a nullary **nil** representing an empty list of premises and a binary **cons**, joining the first premise in a list with the rest of that list. With this, we can represent the pair (p, c) as a general term as well, *e.g.*, by using a binary constructor **rule**. Thus, the rule

[2] We often add a third element to each rule: its name. But such names have no formal meaning. We use them to refer to individual rules during our discussion.

$$\frac{p_1 \qquad \cdots \qquad p_n}{c}$$

becomes

$$\mathbf{rule}(\mathbf{cons}(p_1, \ldots \mathbf{cons}(p_n, \mathbf{nil}) \ldots), c)$$

A rule r' is an *instance* of rule r if there exists a substitution σ such that $r' = \sigma(r)$. (The advantage of viewing rules as terms is that instantiation, which substitutes meta-variables within both premises and conclusions *simultaneously*, becomes nothing more than a particular use of meta-substitution as discussed above.)

1.8 Derivations

Given a set of rules R, a derivation of a term t with respect to R is a *finite* set D of instances of rules in R such that

- $r = \dfrac{p_1 \qquad \cdots \qquad p_k}{t}$ for some premises $p_1, \ldots, p_k$, and
- $D \setminus \{r\}$ is a derivation for each of these p_i.

This definition easily lets us view derivations as trees whose nodes are rule instances as follows: the root of the tree is the r from above. It has k sub-trees where the i-th sub-tree is the derivation for the respective p_i.

Derivations as terms: We can represent derivations trees as terms, too! To see this, let us enhance the above definition of derivations so that it spells out what the term representation of each derivation tree looks like:

Let R be a set of rules. A finite set D of instances of R is a derivation of t and the term d is the term representation of the corresponding derivation tree if and only if there exists a rule R in D such that:

- $r = \mathbf{rule}(\mathbf{cons}(p_1, \ldots \mathbf{cons}(p_k, \mathbf{nil}) \ldots), t)$ for some premises $p_1, \ldots, p_k$, and
- $d = \mathbf{derivation}(\mathbf{cons}(d_1, \ldots, \mathbf{cons}(d_k, \mathbf{nil}) \ldots), t)$ for some terms $d_1, \ldots, d_k$, and
- for each $i \in \{1, \ldots, k\}$ we have that $D \setminus \{r\}$ is a derivation of p_i where d_i is the corresponding derivation tree.

Example: Consider the set R of rules for difference expressions (as terms):

$$\frac{}{\mathbf{zero}\ \mathbf{exp}}\ \text{ZERO} \qquad \frac{}{\mathbf{one}\ \mathbf{exp}}\ \text{ONE} \qquad \frac{e_1\ \mathbf{exp} \qquad e_2\ \mathbf{exp}}{\mathbf{minus}(e_1, e_2)\ \mathbf{exp}}\ \text{MINUS}$$

The following is a visual representation of the derivation for $\mathbf{minus}(\mathbf{one}, \mathbf{zero})\ \mathbf{exp}$. The labels show the name of the original rule together with the substitution (if any) that was used to obtain the particular instance:

$$\dfrac{\dfrac{}{\mathbf{one\ exp}}\,\text{ONE} \qquad \dfrac{}{\mathbf{zero\ exp}}\,\text{ZERO}}{\mathbf{minus}(\mathbf{one}, \mathbf{zero})\ \mathbf{exp}}\,[e_1 \mapsto \mathbf{one}, e_2 \mapsto \mathbf{zero}]\text{MINUS}$$

This corresponds to the following finite set D of instances in R:

$$D = \left\{ \dfrac{}{\mathbf{one\ exp}}\ ,\ \dfrac{}{\mathbf{zero\ exp}}\ ,\ \dfrac{\mathbf{one\ exp} \qquad \mathbf{zero\ exp}}{\mathbf{minus}(\mathbf{one}, \mathbf{zero})\ \mathbf{exp}} \right\}$$

or, as set of terms,

$$D = \left\{ \begin{array}{l} \mathbf{rule}(\mathbf{nil}, \mathbf{one\ exp}),\ \mathbf{rule}(\mathbf{nil}, \mathbf{zero\ exp}), \\ \mathbf{rule}(\mathbf{cons}(\mathbf{one\ exp}, \mathbf{cons}(\mathbf{zero\ exp}, \mathbf{nil})), \mathbf{minus}(\mathbf{one}, \mathbf{zero})\ \mathbf{exp}) \end{array} \right\}$$

Finally, the term d representing the derivation tree is:

$$d = \mathbf{derivation}(\mathbf{cons}(\mathbf{derivation}(\mathbf{nil}, \mathbf{one\ exp}), \mathbf{cons}(\mathbf{derivation}(\mathbf{nil}, \mathbf{zero\ exp}), \mathbf{nil})), \\ \mathbf{minus}(\mathbf{one}, \mathbf{zero})\ \mathbf{exp})$$

1.9 Ground terms vs. general terms

The language of terms and rules is the "meta-language" which we use to talk *about* the object of our study, namely programming languages and their semantics.

All semantic objects correspond to ground terms. In principle, every derivation we use to prove a particular judgment is a ground derivation. So why is it then that we need meta-variables and general terms?

The advantage of general terms is that they represent—in a compact fashion—potentially infinite sets of ground terms. If we were to insist in ground terms only, we would have to write down infinitely large sets of rules and infinitely many derivations. Using meta-variables we can characterize these large sets in finite space by replacing sub-terms by variables.

The key to understanding why this work is the following lemma:

Lemma 1.1
Let R be a set of rules and G be the (usually infinite) set of ground rules obtained by instantiating rules in R. (That is, for each $r' \in G$ there exits a substitution σ and a rule $r \in R$ such that $r' = \sigma(r)$.) If t is a term derivable in R, then every ground instance t' of t is derivable in G.

The proof for this lemma rests on two facts:

1. A derivation is a set of instances of the respective rules, and
2. the "is an instance of" relation is transitive.

The details of the proof are left as an exercise.

1.10 Derivable and admissible rules

Let R be a set of rules. An additional rule r of the form

$$\frac{p_1 \qquad \dots \qquad p_n}{c}$$

where the p_k are the premises and c is the conclusion is called *derivable* in R if there is a derivation for c with respect to $R \cup \{p_1, \dots, p_n\}$. (Notice that the p_k as well as c may contain meta-variables.)

Sometimes we can prove that every ground instance of a rule r is derivable in R while r itself is not derivable. (That is, r can be proved as a "lemma" where the proof usually involves case analysis on the instances of r.) In this case we call r *admissible* in R, meaning that the addition of r to R does not let us derive any additional ground terms, so adding it to R does not change the set of rules substantially.

A rule r that is derivable in R stays derivable in any extension R' of R ($R' \supset R$). However, if r is only admissible, then it usually ceases to be admissible in some extensions of R, *i.e.*, adding certain other rules or axioms to R can invalidate r.

For an example of a derivable and an admissible rule, consider the following set of rules R:

$$\frac{}{\mathbf{zero}\ \mathbf{nat}} \qquad\qquad \frac{n\ \mathbf{nat}}{\mathbf{succ}(n)\ \mathbf{nat}}$$

With this, the rule

$$\frac{n\ \mathbf{nat}}{\mathbf{succ}(\mathbf{succ}(n))\ \mathbf{nat}}$$

is derivable, while

$$\frac{\mathbf{succ}(n)\ \mathbf{nat}}{n\ \mathbf{nat}}$$

is merely admissible. (The proof for this is left as an exercise.)

1.11 Well-founded relations and induction

The mathematical induction principle can be stated as follows:

Proposition 1.2 (mathematical induction)
Let P be a predicate defined over the natural numbers ($\in \mathbb{N}$). If

- *$P(0)$, and also*
- *for any natural n, $P(n)$ implies $P(n+1)$,*

then $P(k)$ holds for all $k \in \mathbb{N}$.

This principle is justified by the following indirect reasoning:

Let $E = \{x \in \mathbb{N} \mid \neg P(x)\}$ be the set of "counterexamples" to the above principle. To say that the principle is false is to say that E is non-empty. In that case let $e_0 = \min(E)$. This e_0 cannot be 0, since we know that $P(0)$ holds. Thus, there must exist an e_1 such that $e_0 = e_1 + 1$. If we assume $P(e_1)$, then we have a contradiction, since the second condition above would dictate that $P(e_0)$ also be true. But if $\neg P(e_1)$, then e_0 was not minimal in E, which is also a contradiction. ■

Well-founded sets: Notice that for the above "proof" of the mathematical induction principle we made use of the *successor* relation between natural numbers. The key insight was that any set of "counterexamples" would have to contain a *minimal* element with respect to the successor relation. We actually picked the minimal element under the natural ordering, but it would have been sufficient to pick some e such that $e - 1$ is not in E. Thus, we made use of the fact that every non-empty set $X \subseteq \mathbb{N}$ contains at least one element x_0 such that $\nexists y \in X.x_0 = y + 1$.

We can generalize this setup by abstracting from the set and its relation and simply insist that whatever set-relation pair (S, R) we use, it be *well-founded*:

Let (S, R) be a pair of a set together with a binary relation over S. Given $X \subseteq S$, an *R-minimal* element m of X is defined to have the following property: $\forall x \in X.\neg xRm$.

Definition 1.3 (well-founded set *a.k.a.* well-founded relation)
The pair (S, R) where S is a set and $R \subseteq S \times S$ is a binary relation over S is well-founded, if

$$\forall X \subseteq S.X \neq \emptyset \Rightarrow \exists x_0 \in X.\ \nexists y \in X.yRx_0.$$

In English: Every non-empty subset of S has at least one R-minimal element.

Well-founded induction: Every well-founded set has an induction principle:

Proposition 1.4 (well-founded induction)
Let (S, R) be a well-founded set and let P be a predicate defined over S. If the following condition is true, then $P(x)$ holds for all $x \in S$:

$$\forall x \in S.(\forall y \in S.yRx \Rightarrow P(y)) \Rightarrow P(x)$$

In English: If the fact that P holds for every element that is "smaller" than x (in the sense of R) implies that P also holds for x, then P is true for all elements of S.

Alternative definition of well-founded sets: In the literature we often find the following alternative definition of well-founded sets. Assuming the *axiom of choice* it is equivalent to the one given above:

Definition 1.5 (well-founded sets (alt. def.))
The set-relation pair (S, R) is well-founded if S does not contain infinitely descending R-chains.

Here, an *infinitely descending chain* is an infinite sequence $x_1, x_2, \ldots$ of elements in S such that $\forall i. x_{i+1} R x_i$.

Examples for well-founded sets:

- The pair $(\mathbb{N}, \mathrm{suc})$ where $suc(x, y)$ iff $x + 1 = y$ is well-founded. The resulting induction principle is that of mathematical induction.
- The pair $(\mathbb{N}, <)$ (natural numbers under their natural ordering) is well-founded. The resulting induction principle is known as *complete induction*.
- Let (S_1, R_1) and (S_2, R_2) be two well-founded sets. Then the pair $(S_1 \times S_2, R)$ where $(x_1, x_2) R (y_1, y_2)$ iff $x_1 R_1 y_1$ and $x_2 R_2 y_2$ is well-founded. The same is true for a variety of other possible ways of defining R in terms of R_1 and R_2. Examples include using just R_1 (or R_2) on the left (or right) projection, and—if R_1 is a strict total order on S_1—the *lexicographic* order: $(x_1, x_2) R (y_1, y_2)$ iff either $x_1 R_1 y_1$ or not $x_1 R_1 y_1$ and also not $y_1 R_1 x_1$ and also $x_2 R_2 y_2$.
- Let S be a set and M be a function from S to $\mathbb{N}$. If we define R as $\{(x, y) \mid M(x) < M(y)\}$, then (S, R) is well-founded. The function M is often called a *metric*.
- More generally, suppose we have a well-founded set (S_0, R_0). Let S be a set and M be a function from S to S_0 and let us define R as $\{(x, y) \mid (M(x), M(y)) \in R_0\}$. Then (S, R) is also well-founded.

2 Lexical scope and Binding structure

We now consider a small language with natural numbers, addition, multiplication, variables, and *local bindings*.

First, we need a way of denoting *variables*. Variables are atomic names, drawn from some countable infinite set. We do not care what that set actually is—as long as we can compare its elements for equality. For example, we can make the set of variables isomorphic to that of the natural numbers:

$$\frac{}{\mathbf{zero}\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathbf{succ}(n)\ \mathbf{nat}} \qquad \frac{n\ \mathbf{nat}}{\mathbf{id}(n)\ \mathbf{var}}$$

Here $\mathbf{id}(n)$ is a **var** whenever n is a **nat**. Many other schemes for encoding variables are possible, and this is the last time we will show a concrete one. In the future we will simply say something like "x is drawn from some countably infinite set of variables." (In fact, often we will not say much at all.)

A simple language of expressions that contains *constants* (which in this case are natural numbers), *variables* (as explained above), *additions* and *multiplications* as well as a local binding form *let* is the following:

$$\frac{n\ \mathbf{nat}}{n\ \mathbf{exp}} \qquad \frac{x\ \mathbf{var}}{x\ \mathbf{exp}} \qquad \frac{e_1\ \mathbf{exp} \qquad e_2\ \mathbf{exp}}{e_1 + e_2\ \mathbf{exp}} \qquad \frac{e_1\ \mathbf{exp} \qquad e_2\ \mathbf{exp}}{e_1 * e_2\ \mathbf{exp}}$$

$$\frac{x\ \mathbf{var} \qquad e_1\ \mathbf{exp} \qquad e_2\ \mathbf{exp}}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}$$

Usually we will abbreviate such a definition using a mix of informal mathematical set notation and BNF:

$$\begin{array}{lcl} n & \in & \mathbb{N} \\ x & \in & \text{variables} \\ e & ::= & n \mid x \mid e + e \mid e * e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \end{array}$$

2.1 Static semantics

Not all expressions that conform to the grammar are actually "good" expressions. We want to reject expressions that have "dangling" references to variables which are not in scope.

The judgment $\Gamma \vdash e\ \mathbf{ok}$ expresses that e is an acceptable expression if it appears in a *context* described by Γ. In this simple case, Γ keeps track of which variables are currently in scope, so we treat it as a set of variables.

An expression is acceptable as a *program* if it is an expression that makes no demands on its context, *i.e.*, $\emptyset \vdash e\ \mathbf{ok}$.

The rules for deriving judgments of the form $\Gamma \vdash e\ \mathbf{ok}$ follow the grammar of the language closely:

$$\frac{}{\Gamma \vdash n\ \mathbf{ok}}\ \textsc{num-ok} \qquad \frac{x \in \Gamma}{\Gamma \vdash x\ \mathbf{ok}}\ \textsc{var-ok} \qquad \frac{\Gamma \vdash e_1\ \mathbf{ok} \qquad \Gamma \vdash e_2\ \mathbf{ok}}{\Gamma \vdash e_1 + e_2\ \mathbf{ok}}\ \textsc{add-ok}$$

$$\frac{\Gamma \vdash e_1\ \mathbf{ok} \qquad \Gamma \vdash e_2\ \mathbf{ok}}{\Gamma \vdash e_1 * e_2\ \mathbf{ok}}\ \textsc{mul-ok} \qquad \frac{\Gamma \vdash e_1\ \mathbf{ok} \qquad \Gamma \cup \{x\} \vdash e_2\ \mathbf{ok}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{ok}}\ \textsc{let-ok}$$

Notice that we are making some additional notational simplifications here:

- We omit the premise n **nat** from the first rule and rely on typographic convention, namely the use of the meta-variable n, which—in accordance with the earlier definition $n \in \mathbb{N}$—is implicitly understood to be drawn from the naturals. This technique is widely used, especially if the grammar is not given in form of derivation rules but employs the more common BNF-style.

- The second rule has a precondition which is not a judgment but rather a statement of set theory. While we could encode sets and operations

on sets as terms, we usually do not go to that much trouble.[3] In the textbook, such non-judgment premises are notationally distinguished by putting them into parentheses.

- In the last rule we use an operation from set theory (union) as part of a judgment.

2.2 Values

When discussing the dynamic semantics of a language, *i.e.*, the way its programs "execute" at run-time, we need to have a grasp on the "result" or "outcome" of a computation. For this purpose we define a subset of terms that we call *values*. As we will see, the "execution" of an expression is a process that (if everything goes well) terminates once it produces such a value.

In the case of our very simple language, values are simply natural numbers:

$$v \quad ::= \quad n$$

2.3 Substitution

To specify the role of variables in our semantics we rely on the notion of *substitution*. Informally, to evaluate a **let**-expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ we first evaluate e_1 to its values v_1 and then substitute v_1 for the variable x in e_2 before proceeding with the evaluation of the result of this substitution.

Since our language supports *nested* lexical scopes, the definition of substitution is slightly more complicated than that of meta-substitutions. To see this, consider substituting the value 3 for x in $\mathbf{let}\ x = x + x\ \mathbf{in}\ x * x$. Of the 5 occurrences of variable x, only 2 have to be replaced with 3. The third occurrence is a *binding* occurrence, which happens to bind the same variable x. Uses of x within this nested scope are independent of uses that occur outside. Thus, the result of the substitution must be $\mathbf{let}\ x = 3 + 3\ \mathbf{in}\ x * x$ instead of $\mathbf{let}\ x = 3 + 3\ \mathbf{in}\ 3 * 3$ (or even $\mathbf{let}\ 3 = 3 + 3\ \mathbf{in}\ 3 * 3$—which would not even be syntactically well-formed anymore).

For the time being we will only consider the substitution of values for variables in expressions. Since values are closed terms, this simplifies the definition of substitution because we do not have to worry about *capture avoidance*, a topic that we will re-visit later.

We write $\{v/x\}e$ to denote the substitution of value v for variable x in expression e. Here is the definition, by induction on the structure of e:

$$\begin{array}{lcll} \{v/x\}n & = & n & \\ \{v/x\}x & = & v & \\ \{v/x\}y & = & y & ;\ x \neq y \end{array}$$

[3] For *mechanization* it is often necessary to do so after all.

$$
\begin{array}{rcll}
\{v/x\}(e_1 + e_2) & = & (\{v/x\}e_1) + (\{v/x\}e_2) & \\
\{v/x\}(e_1 * e_2) & = & (\{v/x\}e_1) * (\{v/x\}e_2) & \\
\{v/x\}(\textbf{let } x = e_1 \textbf{ in } e_2) & = & \textbf{let } x = \{v/x\}e_1 \textbf{ in } e_2 & \\
\{v/x\}(\textbf{let } y = e_1 \textbf{ in } e_2) & = & \textbf{let } y = \{v/x\}e_1 \textbf{ in } \{v/x\}e_2 & ;\ x \neq y
\end{array}
$$

Notice how the last two lines guarantee that substitution for x does not proceed into the scope of a new, nested binding of the same variable x.

2.4 Structural operational semantics

$$\frac{n = n_1 + n_2}{n_1 + n_2 \mapsto^1 n}\ \text{ADD} \qquad \frac{n = n_1 n_2}{n_1 + n_2 \mapsto^1 n}\ \text{MUL} \qquad \frac{}{\textbf{let } x = v \textbf{ in } e \mapsto^1 \{v/x\}e}\ \text{LET}$$

$$\frac{e_1 \mapsto^1 e_1'}{e_1 + e_2 \mapsto^1 e_1' + e_2}\ \text{PLUS-L} \qquad \frac{e_1 \mapsto^1 e_1'}{e_1 * e_2 \mapsto^1 e_1' * e_2}\ \text{TIMES-L}$$

$$\frac{e_2 \mapsto^1 e_2'}{v_1 + e_2 \mapsto^1 v_1 + e_2'}\ \text{PLUS-R} \qquad \frac{e_2 \mapsto^1 e_2'}{v_1 * e_2 \mapsto^1 v_1 * e_2'}\ \text{TIMES-R}$$

$$\frac{e_1 \mapsto^1 e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \mapsto^1 \textbf{let } x = e_1' \textbf{ in } e_2}\ \text{LET-L}$$

Progress and preservation: The operational semantics does not tell us how to evaluate every possible expression. For example, the expression $x + 3$ is "stuck," as no rule applies to it. The problem here is with the free reference to variable x. Intuitively, expressions with free variables are not "complete" and make no sense on their own. To evaluate an expression, we want it to be well-formed—which in our case means we want it to be closed. This is precisely the property expressed by our **ok** judgments, *i.e.*, our very primitive notion of a static semantics.

The following two lemmas connect static and dynamic semantics and together establish the property called *safety*:

Lemma 2.1 (preservation)
If $\emptyset \vdash e$ **ok** *and* $e \mapsto^1 e'$*, then* $\emptyset \vdash e'$ **ok**.

Lemma 2.2 (progress)
If $\emptyset \vdash e$ **ok** *and* e *is not a value, then there exists an* e' *such that* $e \mapsto^1 e'$.

Proof of preservation: The proof proceeds by induction on the structure of e—or, equivalently, on the structure of the derivation for $\emptyset \vdash e$ **ok**. We perform a big case analysis, considering each of the evaluation rules that could have been used last when deriving $e \mapsto^1 e'$. The reasoning for most of these rules falls within two patterns:

1. If the rule in question describes a computation step, the result follows immediately. Example: If rule ADD was used, then the result has the form n so that rule NUM-OK applies, which yields the desired conclusion, namely $\emptyset \vdash n$ **ok**.

2. If the rule in question is a structural rule, the result follows by a simple application of the induction hypothesis (IH). Example: If rule PLUS-L was used, then e must have the form $e_1 + e_2$ and we have $e_1 \mapsto^1 e_1'$. By inversion of ADD-OK we have $\emptyset \vdash e_1$ **ok**, so the IH applies, giving us $\emptyset \vdash e_1'$ **ok**. Using rule ADD-OK in forward direction yields the desired conclusion, namely that $\emptyset \vdash e_1' + e_2$ **ok**.

The only tricky rule is LET. Here we need to argue somehow that given some n and some e with $\{x\} \vdash e$ **ok** we have $\emptyset \vdash \{n/x\}e$ **ok**. We separate this statement out as a so-called *substitution lemma*, which we discuss below. ■

The substitution lemma: Roughly speaking, the substitution lemma states that substituting a value into a well-formed term keeps it well-formed. Concretely, we need that if e has only one free variable x, then substituting some value n for x in e yields a well-formed, closed expression.

The proof for this lemma is by induction on the structure of e. However, a naive attempt at this proof fails for the case of **let** $y = e_1$ **in** e_2, because the induction hypothesis does not hold for e_2 due to the fact that e_2 is allowed to make free references to y in addition to x, so it is not necessarily the case that $\{x\} \vdash e_2$ **ok** as we only know that $\{x, y\} \vdash e_2$ **ok**. To make the proof work, we need to *strengthen* the induction hypothesis, which effectively means we are proving a stronger lemma:

Lemma 2.3 (substitution)
Let n be a natural. Further, let $x \notin \Gamma$ and $\Gamma \cup \{x\} \vdash e$ **ok**. *Then $\Gamma \vdash \{n/x\}e$* **ok**.

The proof of this stronger version of the substitution lemma is left as an exercise. ■

Proof of progress: The proof of progress also proceeds by induction on the derivation of $\emptyset \vdash e$ **ok**. It performs a case analysis on e's head constructor and argues that in each case (except when $e = n$, at which point we have a value) it can take at least one step. The details are left as an exercise. ■

Safety: *Preservation* states that a well-formed, closed expression stays well-formed and closed when it takes a step. *Progress* states that a well-formed, closed expression that is not already a value can take (at least) one step. Together the two imply that starting from a well-formed, closed expression, one will either reach a value after a finite number of steps or otherwise be able to take infinitely many steps without ever "getting stuck." This property of a language is often called *safety*.

To formalize the concept of taking finitely many steps we take the *transitive closure* of $\mapsto^1$. The result is a *transition system*, which describes the process of taking a well-formed closed term e (*i.e.*, $\emptyset \vdash e$ **ok**) through a sequence of

individual steps. The transitive closure is the smallest relation that is both *reflexive* and *closed under head expansion*:

$$\frac{}{e \mapsto^* e}\ \text{REFL} \qquad\qquad \frac{e \mapsto^1 e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}\ \text{CUHE}$$

The restriction of $\mapsto^*$ to result terms that are values represents the *small step semantics* of our language.

2.5 Evaluation semantics

An alternative form of semantics dispenses with individual "single" steps and describes the entire evaluation process as one "big" step. We write $e \Downarrow v$ to say that e *evaluates to* v.

The rules for deriving judgments of this form are the following:

$$\frac{}{n \Downarrow n} \qquad\qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 n_2}{e_1 * e_2 \Downarrow n} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad \{v_1/x\}e_2 \Downarrow v_2}{\textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow v_2}$$

As you can see, the evaluation semantics appears to be significantly more compact than the small-step semantics. In particular, it does not need any of the structural rules that in the small-step case are used to direct the focus of the evaluation to the *current redex*. (A *redex* is a *reducible expression*, *i.e.*, an expression to which a computation rule applies. If a program that is not already a value contains multiple redexes, then one of them is the one that will be reduced next. That redex is called the *current redex*, and the structural rules of a small-step semantics specify how to find it.)

A big-step semantics is less explicit about how given an e a mechanical process would actually find the corresponding v such that $e \Downarrow v$. In particular, the rules we gave do not say in which order the subterms of $+$ and $*$ should be evaluated.

For the given language, it turns out that this order does not matter. One way of proving this is to establish that $\Downarrow$ and $\mapsto^*$ represent the same relation between well-formed expressions and values.

Lemma 2.4 (Big- and small-step semantics are equivalent)
If $\emptyset \vdash e$ **ok** *then* $e \Downarrow v$ *if and only if* $e \mapsto^* v$.

In general, whenever we use several different ways of giving a semantics to what is intended to be the same language, it is our obligation to prove that these semantics coincide.

2.6 Why so many styles?

Why are there so many styles of defining a semantics? The answer is that certain properties of languages are easier to prove using one style than another. Other semantics are easier to be turned into working implementations. (There are also reasons of history and tradition, but those are secondary.)

Safety proofs—using small-step semantics

Proving (type-)safety via progress and preservation has become more or less standard. But the very statement of both progress and preservation refers to a small-step semantics. Why is that? Consider this attempt at a safety lemma:

> If $\emptyset \vdash e$ **ok**, then $\exists v.e \Downarrow v$.

Why do we not state and prove this instead?

The reason that we do not do this is that for many practical languages, in particular all languages that are *Turing-complete*, the statement is not true! If programs do not necessarily terminate, they can be safe without actually producing a result. Safety does not express that a program always completes is execution; it merely states that execution will never get stuck unless the execution is completed and has produced a value. Progress- and preservation lemmas do not have this problem, since they do not talk about complete executions but only about one single step along the way.

Of course, it is not completely impossible to deal with this problem in a big-step setting. The standard approach is to explicitly augment the big-step semantics with an "error result," often called *wrong*. Here are the rules:

$$\frac{}{n \Downarrow n} \qquad \frac{}{x \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow \text{wrong}}{e_1 + e_2 \Downarrow \text{wrong}} \qquad \frac{e_2 \Downarrow \text{wrong}}{e_1 + e_2 \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n = n_1 n_2}{e_1 * e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow \text{wrong}}{e_1 * e_2 \Downarrow \text{wrong}} \qquad \frac{e_2 \Downarrow \text{wrong}}{e_1 * e_2 \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow v_1 \qquad \{v_1/x\}e_2 \Downarrow v_2}{\textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \text{wrong}}{\textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow \text{wrong}}$$

You notice that the semantics is suddenly littered with additional error rules that essentially do nothing but only serve to deal with what would normally amount to "stuck" executions. We now have to prove that the so defined relation $\Downarrow$ is complete, *i.e.*, that for all expressions e (well-formed or otherwise) we have that either there is a v such that $e \Downarrow v$ or $e \Downarrow$ wrong. We can then state and prove the safety result, namely: if $\emptyset \vdash e$ **ok**, then $\neg(e \Downarrow \text{wrong})$.

In any case, the chief advantage of the big-step semantics, namely its conciseness, has gone out the window.

3 The Simply-Typed λ-Calculus

We will now gradually move on to more interesting languages that have more than one kind of data. First, we might add boolean values **true** and **false**, together with an **if** form for distinguishing between them. This leaves us with two kinds of data and the situation where a program can become stuck not only because of free variables but also because of operations being applied to data of the wrong *type*. For now we just have two types (call them **nat** and **bool**), but the situation gets worse once we also consider arbitrary *function values*, which give rise to an infinite universe of types.

3.1 Abstracting over free variables—the λ-notation

Our semantics does not directly assign a "meaning" to any expression with free variables. However, by way of substitution, if e has a free variable x, we can see it as mapping any value v to $\{v/x\}e$.

In the λ-calculus (of any flavor), the corresponding "function" has a manifest representation as a syntactic value called an *abstraction*. In the *untyped* λ-calculus, this value is written $\lambda x.e$. For example $\lambda x.x + 1$ is a function of one argument. Informally, it maps this argument to its successor. The entire λ-term is *closed*, the λ-symbol *binds* variable x within the body e, much like **let** $x = e$ **in** e' binds x within e'.

If there is more than one free variable, we can iterate the process of forming abstractions: $\lambda x.\lambda y.x + y$ is a function taking one argument and producing another function, which itself takes a second argument and finally produces the sum of both arguments. Contrast this with $\lambda x.x + y$, which still has a free variable y.

3.2 Abstract syntax

types:	τ	$::=$	**nat** $\mid$	nat. numbers
			bool $\mid$	booleans
			$\tau \to \tau$	functions
values:	v	$::=$	$n \mid$	numbers (intro. **nat**)
			false $\mid$ **true** $\mid$	booleans (intro. **bool**)
			$\lambda x : \tau.e$	λ-abstraction (intro. $\tau_1 \to \tau_2$)
expressions:	e	$::=$	$x \mid$	variables
			$v \mid$	values
			pred(e) $\mid$ **succ**(e) $\mid$ **iszero**(e) $\mid$	arithmetic (elim. **nat**)
			if e **then** e **else** $e \mid$	conditional (elim. **bool**)
			$e\ e$	appliation (elim. $\tau_1 \to \tau_2$)

3.3 Static semantics: typing rules

Typing judgment: $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \mathbf{nat}}\ \text{I-NAT} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}\ \text{I-BOOL(T)} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}\ \text{I-BOOL(F)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \text{I}_{\to} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \mathbf{nat}}{\Gamma \vdash \mathbf{pred}(e) : \mathbf{nat}}\ \text{E-NAT(P)}$$

$$\frac{\Gamma \vdash e : \mathbf{nat}}{\Gamma \vdash \mathbf{succ}(e) : \mathbf{nat}}\ \text{E-NAT(S)} \qquad \frac{\Gamma \vdash e : \mathbf{nat}}{\Gamma \vdash \mathbf{iszero}(e) : \mathbf{bool}}\ \text{E-NAT(Z)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ \text{E-BOOL}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{E}_{\to}$$

3.4 Small-step semantics

Single step relation: $\boxed{e \mapsto^1 e'}$

$$\frac{n' = n - 1}{\mathbf{pred}(n) \mapsto^1 n'} \qquad \frac{n' = n + 1}{\mathbf{succ}(n) \mapsto^1 n'} \qquad \frac{}{\mathbf{iszero}(0) \mapsto^1 \mathbf{true}}$$

$$\frac{n \neq 0}{\mathbf{iszero}(n) \mapsto^1 \mathbf{false}} \qquad \frac{}{\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 \mapsto^1 e_1}$$

$$\frac{}{\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 \mapsto^1 e_2} \qquad \frac{}{(\lambda x : \tau . e)\ v \mapsto^1 \{v/x\}e}$$

$$\frac{e \mapsto^1 e'}{\mathbf{pred}(e) \mapsto^1 \mathbf{pred}(e')} \qquad \frac{e \mapsto^1 e'}{\mathbf{succ}(e) \mapsto^1 \mathbf{succ}(e')} \qquad \frac{e \mapsto^1 e'}{\mathbf{iszero}(e) \mapsto^1 \mathbf{iszero}(e')}$$

$$\frac{e_1 \mapsto^1 e_1'}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mapsto^1 \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3} \qquad \frac{e_1 \mapsto^1 e_1'}{e_1\ e_2 \mapsto^1 e_1'\ e_2}$$

$$\frac{e_2 \mapsto^1 e_2'}{v_1\ e_2 \mapsto^1 v_1\ e_2'}$$

3.5 Big-step semantics

Evaluation relation: $\boxed{e \Downarrow v}$

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow n \qquad n' = n - 1}{\mathbf{pred}(e) \Downarrow n'} \qquad \frac{e \Downarrow n \qquad n' = n + 1}{\mathbf{succ}(e) \Downarrow n'}$$

$$\frac{e \Downarrow 0}{\mathbf{iszero}(e) \Downarrow \mathbf{true}} \qquad \frac{e \Downarrow n \qquad n \neq 0}{\mathbf{iszero}(e) \Downarrow \mathbf{false}} \qquad \frac{e_1 \Downarrow \mathbf{true} \qquad e_2 \Downarrow v_2}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \mathbf{false} \qquad e_3 \Downarrow v_3}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_3} \qquad \frac{e_1 \Downarrow \lambda x : \tau_2.e \qquad e_2 \Downarrow v_2 \qquad \{v_2/x\}e \Downarrow v}{e_1\ e_2 \Downarrow v}$$

3.6 Making control explicit

The small-step semantics and the big-step semantics each suffer from a certain "defect" related to the problem of finding the *current redex*, *i.e.*, the subexpression that where the next computation step will happen. This defect makes them less suitable as models of how computation in a real computer progresses:

small-step: In the small-step semantics we have a subset of the rules (the *structural* or *search* rules) dealing exclusively with the problem of focusing on the current redex. The problem is that the search for the current redex is done over and over again, each time starting at the top, with the whole program. In reality, programs do not execute in such a convoluted fashion. Instead, real programs execute as a sequence of computation steps which are interspersed with certain small administrative steps (*e.g.*, saving intermediate results, postponing a sub-computation until another sub-computation is complete, *etc.*).

big-step: In the big-step formulation the management of the current redex is left entirely implicit. In the big-step semantics given above, flow of control is not specified. Indeed, unlike in the small-step version, there is not even any indication of whether evaluation proceeds from left to right or from right to left. The big-step semantics does not distinguish individual *states* and does not give immediate rise to a notion of computation by repeated state transition.

To fix these defects is to model the interplay of computation steps and administrative steps explicitly. The result is an *abstract machine*. In this section we describe an abstract machine that makes *control* explicit by representing it as a *stack* of *frames*. Each frame corresponds to a piece of work that has been postponed until a sub-computation is complete.

Our first machine, which we called the *C machine*[4], consists of states (k, e) (where k is a stack and e is the *current expression*) and a *transition* relation between states. To express that k is a stack we use the judgment k **stack**.

[4]because it represents $\underline{C}$ontrol explicitly

Similarly, to express that f is a frame we say f **frame**. The empty stack is $\bullet$, a frame f on top of stack k is written $f \rhd k$:

$$\frac{}{\bullet\ \textbf{stack}} \qquad \frac{f\ \textbf{frame} \qquad k\ \textbf{stack}}{f \rhd k\ \textbf{stack}}$$

Each frame corresponds to one of the search rules of the small-step semantics (assuming e **exp** and v **val** judgments for expressions and values, respectively):

$$\frac{}{\textbf{pred}(\square)\ \textbf{frame}} \qquad \frac{}{\textbf{succ}(\square)\ \textbf{frame}} \qquad \frac{}{\textbf{iszero}(\square)\ \textbf{frame}}$$

$$\frac{e_2\ \textbf{exp} \qquad e_3\ \textbf{exp}}{\textbf{if}\ \square\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{frame}} \qquad \frac{e_2\ \textbf{exp}}{\square\ e_2\ \textbf{frame}} \qquad \frac{v_1\ \textbf{val}}{v_1\ \square\ \textbf{frame}}$$

Alternatively, we might set this up using BNF-style declarations for f and k:

$$\begin{array}{lcl} f & ::= & \textbf{pred}(\square) \mid \textbf{succ}(\square) \mid \textbf{iszero}(\square) \mid \textbf{if}\ \square\ \textbf{then}\ e\ \textbf{else}\ e \mid \square\ e \mid v\ \square \\ k & ::= & \bullet \mid f \rhd k \end{array}$$

The machine semantics is now given as a set of single-step transition rules[5] $(k, e) \mapsto_{\mathrm{C}} (k', e')$ between states:

$$\begin{array}{rcll} (k, \textbf{pred}(e)) & \mapsto_{\mathrm{C}} & (\textbf{pred}(\square) \rhd k, e) & \\ (\textbf{pred}(\square) \rhd k, n) & \mapsto_{\mathrm{C}} & (k, n') & ; n' = n - 1 \\ (k, \textbf{succ}(e)) & \mapsto_{\mathrm{C}} & (\textbf{succ}(\square) \rhd k, e) & \\ (\textbf{succ}(\square) \rhd k, n) & \mapsto_{\mathrm{C}} & (k, n') & ; n' = n + 1 \\ (k, \textbf{iszero}(e)) & \mapsto_{\mathrm{C}} & (\textbf{iszero}(\square) \rhd k, e) & \\ (\textbf{iszero}(\square) \rhd k, 0) & \mapsto_{\mathrm{C}} & (k, \textbf{true}) & \\ (\textbf{iszero}(\square) \rhd k, n) & \mapsto_{\mathrm{C}} & (k, \textbf{false}) & ;\ n \neq 0 \\ (k, \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & \mapsto_{\mathrm{C}} & (\textbf{if}\ \square\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rhd k, e_1) & \\ (\textbf{if}\ \square\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rhd k, \textbf{true}) & \mapsto_{\mathrm{C}} & (k, e_2) & \\ (\textbf{if}\ \square\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rhd k, \textbf{false}) & \mapsto_{\mathrm{C}} & (k, e_3) & \\ (k, e_1\ e_2) & \mapsto_{\mathrm{C}} & (\square\ e_2 \rhd k, e_1) & \\ (\square\ e_2 \rhd k, v) & \mapsto_{\mathrm{C}} & (v\ \square \rhd k, e_2) & \\ ((\lambda x : \tau . e)\ \square \rhd k, v) & \mapsto_{\mathrm{C}} & (k, \{v/x\}e) & \end{array}$$

[5] None of the rules have any premises that refer to other instances of the transition relation, so we omit the horizontal bars write down only the conclusions of these rules. In the few cases where a rule needs a simple side condition, the condition is given "on the side," separated by a semicolon ";".

As described in the textbook, we can organize these transition rules somewhat differently to make it easier to compare to the environment-based machine (*E machine*) semantics that we will see shortly. For this, we distinguish between two kinds of states:

expression states: A state of the form (k, e) represents the situation where the *current expression* is e. In other words, the machine is about to inspect e in order to determine how to go about processing it. Any previously postponed work is described by the current stack k.

value states: A state of the form (v, k) represents the situation where a previous computation has yielded a value v and the machine is now going to inspect the top of the stack k to see what suspended work has to be resumed. In particular, if k is the empty stack, then our machine has terminated and v is the final result.

The rules for this style of machine are essentially the same rules as above—only the notation of some of the states has changed. There is one extra rule, namely the one for state (k, e) where e happens to be a value:

$$
\begin{array}{rcll}
(k, v) & \mapsto_{\mathrm{C}} & (v, k) & \\
(k, \mathbf{pred}(e)) & \mapsto_{\mathrm{C}} & (\mathbf{pred}(\square) \rhd k, e) & \\
(k, \mathbf{succ}(e)) & \mapsto_{\mathrm{C}} & (\mathbf{succ}(\square) \rhd k, e) & \\
(k, \mathbf{iszero}(e)) & \mapsto_{\mathrm{C}} & (\mathbf{iszero}(\square) \rhd k, e) & \\
(k, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & \mapsto_{\mathrm{C}} & (\mathbf{if}\ \square\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd k, e_1) & \\
(k, e_1\ e_2) & \mapsto_{\mathrm{C}} & (\square\ e_2 \rhd k, e_1) & \\
(n, \mathbf{pred}(\square) \rhd k) & \mapsto_{\mathrm{C}} & (n', k) & ; n' = n - 1 \\
(n, \mathbf{succ}(\square) \rhd k) & \mapsto_{\mathrm{C}} & (n', k) & ; n' = n + 1 \\
(0, \mathbf{iszero}(\square) \rhd k) & \mapsto_{\mathrm{C}} & (\mathbf{true}, k) & \\
(n, \mathbf{iszero}(\square) \rhd k) & \mapsto_{\mathrm{C}} & (\mathbf{false}, k) & ;\ n \neq 0 \\
(\mathbf{true}, \mathbf{if}\ \square\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd k) & \mapsto_{\mathrm{C}} & (k, e_2) & \\
(\mathbf{false}, \mathbf{if}\ \square\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd k) & \mapsto_{\mathrm{C}} & (k, e_3) & \\
(v, \square\ e_2 \rhd k) & \mapsto_{\mathrm{C}} & (v\ \square \rhd k, e_2) & \\
(v, (\lambda x : \tau.e)\ \square \rhd k) & \mapsto_{\mathrm{C}} & (k, \{v/x\}e) &
\end{array}
$$

3.7 Environments

An alternative version of our big-step semantics does not use substitution directly. Instead, the substitution that is supposed to happen at the time of function application is "remembered" in a data-structure called an *environment* and applied later, when the value of a variable is actually needed. In effect, substitutions happen lazily.

Since we are not performing substitutions directly, we will encounter open expressions and open values (expressions and values that have free variables). Dealing with variables is relatively easy, as we can simply look them up in the current environment. Primitive values such as numbers n and boolean values **true** and **false** are closed and need no special treatment. But values that result from the evaluation of open λ-expressions must remember the environment that was in effect at the time the expression was evaluated.

This observation gives rise to the notion of *machine* values V (*a.k.a.*, *internal* values), which the semantics uses to represent the results of evaluations. In particular, evaluating the (possibly open) λ-term L of the form $\lambda x : \tau.e$ in the environment E results in the pair $\langle L, E\rangle$. Such a pair is called a *closure*.

Environments are finite mappings from variables to machine values:

$$\begin{array}{lcl} V & ::= & n \mid \mathbf{true} \mid \mathbf{false} \mid \langle \lambda x : \tau.e, E\rangle \\ E & \in & \mathrm{Var} \mapsto^{\mathrm{fin}} V \end{array}$$

Evaluation relation: $\boxed{(E, e) \Downarrow V}$

$$\frac{E(x) = V}{(E, x) \Downarrow V} \qquad \frac{}{(E, n) \Downarrow n} \qquad \frac{}{(E, \mathbf{true}) \Downarrow \mathbf{true}} \qquad \frac{}{(E, \mathbf{false}) \Downarrow \mathbf{false}}$$

$$\frac{}{(E, \lambda x : \tau.e) \Downarrow \langle \lambda x : \tau.e, E\rangle} \qquad \frac{(E, e) \Downarrow n \qquad n' = n - 1}{(E, \mathbf{pred}(e)) \Downarrow n'}$$

$$\frac{(E, e) \Downarrow n \qquad n' = n + 1}{(E, \mathbf{succ}(e)) \Downarrow n'} \qquad \frac{(E, e) \Downarrow 0}{(E, \mathbf{iszero}(e)) \Downarrow \mathbf{true}} \qquad \frac{(E, e) \Downarrow n \qquad n \neq 0}{(E, \mathbf{iszero}(e)) \Downarrow \mathbf{false}}$$

$$\frac{(E, e_1) \Downarrow \mathbf{true} \qquad (E, e_2) \Downarrow V_2}{(E, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \Downarrow V_2} \qquad \frac{(E, e_1) \Downarrow \mathbf{false} \qquad (E, e_3) \Downarrow V_3}{(E, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \Downarrow V_3}$$

$$\frac{(E, e_1) \Downarrow \langle \lambda x : \tau_2.e, E'\rangle \qquad (E, e_2) \Downarrow V_2 \qquad (E'[x \mapsto V_2], e) \Downarrow V}{(E, e_1\ e_2) \Downarrow V}$$

The *E machine* is an abstract machine that corresponds to the environment-based semantics in the same way in which the C machine corresponds to the substitution-based semantics. Of course, the E machine uses *machine values* where the C machine uses ordinary values. We use capital letters F to distinguish E machine frames from C machine frames f. Similarly, the E machine's stacks are written K. Value states have the form (V, K), and expression states have the form (K, E, e) where E is the *current environment*.

When the evaluation of an expression is postponed by placing it on the stack, the current environment has to be saved along with it. Therefore, the language of frames is:

$$
\begin{array}{lcl}
F & ::= & \mathbf{pred}(\Box) \mid \mathbf{succ}(\Box) \mid \mathbf{iszero}(\Box) \mid \langle \mathbf{if}\ \Box\ \mathbf{then}\ e\ \mathbf{else}\ e, E\rangle \mid \langle \Box\ e, E\rangle \mid V\ \Box \\
K & ::= & \bullet \mid F \rhd K
\end{array}
$$

With this preparation, the set of transition rules for the E machine now follows the example of the C machine quite closely:

$$
\begin{array}{rcll}
(K, E, n) & \mapsto_{\mathrm{E}} & (n, K) & \\
(K, E, \mathbf{true}) & \mapsto_{\mathrm{E}} & (\mathbf{true}, K) & \\
(K, E, \mathbf{false}) & \mapsto_{\mathrm{E}} & (\mathbf{false}, K) & \\
(K, E, \lambda x : \tau.e) & \mapsto_{\mathrm{E}} & (\langle \lambda x : \tau.e, E\rangle, K) & \\
(K, E, \mathbf{pred}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{pred}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{succ}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{succ}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{iszero}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{iszero}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & \mapsto_{\mathrm{E}} & (\langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E\rangle \rhd K, E, e_1) & \\
(K, E, e_1\ e_2) & \mapsto_{\mathrm{E}} & (\langle \Box\ e_2, E\rangle \rhd K, E, e_1) & \\
(n, \mathbf{pred}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (n', K) & ; n' = n - 1 \\
(n, \mathbf{succ}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (n', K) & ; n' = n + 1 \\
(0, \mathbf{iszero}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (\mathbf{true}, K) & \\
(n, \mathbf{iszero}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (\mathbf{false}, K) & ;\ n \neq 0 \\
(\mathbf{true}, \langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E\rangle \rhd K) & \mapsto_{\mathrm{E}} & (K, E, e_2) & \\
(\mathbf{false}, \langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E\rangle \rhd K) & \mapsto_{\mathrm{E}} & (K, E, e_3) & \\
(V, \langle \Box\ e_2, E\rangle \rhd K) & \mapsto_{\mathrm{E}} & (V\ \Box \rhd K, E, e_2) & \\
(V, \langle \lambda x : \tau.e, E\rangle\ \Box \rhd K) & \mapsto_{\mathrm{E}} & (K, E[x \mapsto V], e) &
\end{array}
$$

Observe how the machine carefully maintains the invariant that at the time of any transition of the form $(V, K) \mapsto_{\mathrm{E}} (K', E, e)$ it finds the relevant information for constructing E either within V (namely in the case of function application) or within the top frame of K.

Another interesting detail is in the transition from a state of the form (K, E, v) to a corresponding state (V, K). Unlike in the C machine, where we simply went from (k, v) to (v, k) we now have to transform the syntactic value v to a machine value V. For values of base type this mapping is the identity, but for λ-expressions it involves constructing a closure by pairing the expression with its environment E.

4 Continuations—Controlling Control

In both the C and the E machines, the notion of "control," *i.e.*, the aspect of the semantics that determines what part of the overall work is to be done

when, is made explicit in the control stack (k for the C machine, K for the E machine). However, either machine treats this stack strictly as an *ephemeral* data structure: after a frame has been popped off the stack, it is never accessed again.

It is particularly revealing to compare the treatment of the environment E with that of the stack K in the E machine. Whereas the environment is placed into the stack and—more importantly—into machine values, the stack is kept in only one place and no older version of it survives the moment at which its topmost frame is removed.

This treatment of the control stack is useful if we are interested in an efficient implementation on contemporary hardware, since hardware has been optimized for the case of such an ephemeral stack data structure.[6]

However, in our abstract machine the stack is actually represented by a list, *i.e.*, a data structure that is naturally *persistent*, meaning that new versions can co-exist with old versions in unrestricted ways. Thus, there is no technical problem in devising language features with semantics that amount to manipulating the control stack. In particular, we can add two complementary expression forms that let us

- *capture* the current control stack, and
- *reinstate* a previously captured stack, thereby *throwing away* the current control stack.

In the C and E machines, the control stack in a state (k, e) or (K, E, e) represents the "future of the computation," *i.e.*, all the pending work that has to be accomplished after the evaluation of e is complete. We call this future computation the *continuation* of e. In other words, the control stack is how the machine represents the *current continuation*.

In keeping with the above plan, we add the following two expression forms:

letcc $x : \tau$ **cont in** e**:** This form evaluates e in the scope of the variable x of type τ **cont** which will be bound to a value representing the current continuation. The type of the overall expression is τ.

throw e_1 **to** $e_2 : \tau'$**:** This form evaluates e_1 to v_1 and e_2 to v_2. The type of e_2 must be τ **cont** where τ is the type of e_1. Evaluating this form will not return a value to the current continuation. Instead, the current continuation is discarded and the continuation represented by v_2 is reinstated as the new current continuation and v_1 is returned to it. Since evaluation does not return normally, the type of the expression can be arbitrarily chosen. To keep our typing judgments single-valued, we require this (arbitrary) type to be specified as τ'.

[6] It is usually implemented as a contiguous region of memory with a dedicated register—the *stack pointer*—marking location of the top frame. Frames are added and removed simply by decrementing or incrementing the value of the stack pointer register. Memory occupied by an old frame is overwritten once a new frame has been allocated in its place.

Formally, we augment our language as follows:

types:	τ	$::=$	$\mathbf{nat} \mid$	nat. numbers
			$\mathbf{bool} \mid$	booleans
			$\tau \to \tau \mid$	functions
			$\tau\ \mathbf{cont}$	continuations
values:	v	$::=$	$n \mid$	numbers (intro. **nat**)
			$\mathbf{false} \mid \mathbf{true} \mid$	booleans (intro. **bool**)
			$\lambda x : \tau.e$	λ-abstraction (intro. $\tau_1 \to \tau_2$)
expressions:	e	$::=$	$x \mid$	variables
			$v \mid$	values
			$\mathbf{pred}(e) \mid \mathbf{succ}(e) \mid \mathbf{iszero}(e) \mid$	arithmetic (elim. **nat**)
			$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid$	conditional (elim. **bool**)
			$e\ e \mid$	appliation (elim. $\tau_1 \to \tau_2$)
			$\mathbf{letcc}\ x : \tau\ \mathbf{cont}\ \mathbf{in}\ e \mid$	capture current continuation
			$\mathbf{throw}\ e\ \mathbf{to}\ e : \tau$	reinstate captured continuation

Notice that we did not add any new value forms. In fact, we do not have any syntactic representation of continuations. Such a setup is not compatible with the approach taken by the C machine or any other substitution-based semantics because those represent all runtime values as syntactic values. However, it is easy to modify the E machine by adding another kind of machine value. Doing so is somewhat more realistic since most languages with continuations indeed do not have syntax for representing continuation values.

Here are the typing rules for the extra constructs:

$$\frac{\Gamma, x : \tau\ \mathbf{cont} \vdash e : \tau'}{\Gamma \vdash \mathbf{letcc}\ x : \tau\ \mathbf{cont}\ \mathbf{in}\ e : \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathbf{cont}}{\Gamma \vdash (\mathbf{throw}\ e_1\ \mathbf{to}\ e_2 : \tau') : \tau'}$$

The new definition of machine values V refers to stacks K—resulting in a setup where V, E, F, and K have to be defined simultaneously:

$$\begin{array}{lcl} V & ::= & n \mid \mathbf{true} \mid \mathbf{false} \mid \langle \lambda x : \tau.e, E \rangle \mid K \\ E & \in & \mathrm{Var} \mapsto^{\mathrm{fin}} V \\ F & ::= & \mathbf{pred}(\square) \mid \mathbf{succ}(\square) \mid \mathbf{iszero}(\square) \mid \\ & & \langle \mathbf{if}\ \square\ \mathbf{then}\ e\ \mathbf{else}\ e, E \rangle \mid \langle \square\ e, E \rangle \mid V\ \square \\ & & \langle \mathbf{throw}\ \square\ \mathbf{to}\ e, E \rangle \mid \mathbf{throw}\ V\ \mathbf{to}\ \square \\ K & ::= & \bullet \mid F \rhd K \end{array}$$

The rest of the machine consists of all the transitions of the original E machine, plus those that deal with new expression forms, plus those that deal with new frames:

$$(K, E, n) \quad \mapsto_{\mathrm{E}} \quad (n, K)$$

$$
\begin{array}{rcll}
(K, E, \mathbf{true}) & \mapsto_{\mathrm{E}} & (\mathbf{true}, K) & \\
(K, E, \mathbf{false}) & \mapsto_{\mathrm{E}} & (\mathbf{false}, K) & \\
(K, E, \lambda x : \tau.e) & \mapsto_{\mathrm{E}} & (\langle \lambda x : \tau.e, E \rangle, K) & \\
(K, E, \mathbf{pred}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{pred}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{succ}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{succ}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{iszero}(e)) & \mapsto_{\mathrm{E}} & (\mathbf{iszero}(\Box) \rhd K, E, e) & \\
(K, E, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & \mapsto_{\mathrm{E}} & (\langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E \rangle \rhd K, E, e_1) & \\
(K, E, e_1\ e_2) & \mapsto_{\mathrm{E}} & (\langle \Box\ e_2, E \rangle \rhd K, E, e_1) & \\
(K, E, \mathbf{letcc}\ x : \tau\ \mathbf{cont}\ \mathbf{in}\ e) & \mapsto_{\mathrm{E}} & (K, E[x \mapsto K], e) & \\
(K, E, \mathbf{throw}\ e_1\ \mathbf{to}\ e_2 : \tau) & \mapsto_{\mathrm{E}} & (\langle \mathbf{throw}\ \Box\ \mathbf{to}\ e_2, E \rangle \rhd K, E, e_1) & \\
(n, \mathbf{pred}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (n', K) & ; n' = n - 1 \\
(n, \mathbf{succ}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (n', K) & ; n' = n + 1 \\
(0, \mathbf{iszero}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (\mathbf{true}, K) & \\
(n, \mathbf{iszero}(\Box) \rhd K) & \mapsto_{\mathrm{E}} & (\mathbf{false}, K) & ;\ n \neq 0 \\
(\mathbf{true}, \langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E \rangle \rhd K) & \mapsto_{\mathrm{E}} & (K, E, e_2) & \\
(\mathbf{false}, \langle \mathbf{if}\ \Box\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, E \rangle \rhd K) & \mapsto_{\mathrm{E}} & (K, E, e_3) & \\
(V, \langle \Box\ e_2, E \rangle \rhd K) & \mapsto_{\mathrm{E}} & (V\ \Box \rhd K, E, e_2) & \\
(V, \langle \lambda x : \tau.e, E \rangle\ \Box \rhd K) & \mapsto_{\mathrm{E}} & (K, E[x \mapsto V], e) & \\
(V, \langle \mathbf{throw}\ \Box\ \mathbf{to}\ e_2, E \rangle \rhd K) & \mapsto_{\mathrm{E}} & (\mathbf{throw}\ V\ \mathbf{to}\ \Box \rhd K, E, e_2) & \\
(K', \mathbf{throw}\ V\ \mathbf{to}\ \Box \rhd K) & \mapsto_{\mathrm{E}} & (V, K') &
\end{array}
$$

The last rule is the rule of most interest here: it discards its current continuation K and reinstates a continuation K' that was computed as the value of the second sub-term of a **throw** expression. The other interesting rule is that for **letcc**: it binds the current continuation to x and proceeds with evaluating the expression that is in the scope of x.

4.1 Evaluation order: left-to-right vs. right-to-left

As we have explained before, our big-step semantics do not specify whether composite terms (application of binary operators $e_1 \oplus e_2$ and function application $e_1\ e_2$) evaluate their constituent sub-expressions (e_1 and e_2) from left to right or from right to left. On the other hand, the small-step semantics is explicit about this detail.

In STLC[7] and in MinML (as discussed so far), the difference between left-to-right and right-to-left evaluation orders is not observable. STLC is completely pure, and MinML has only a single effect, namely non-termination. Without effects, order of evaluation is clearly unobservable. With non-termination as the only effect, it is still unobservable, because a non-terminating sub-computation

[7] the Simply Typed λ-calculus

causes overall non-termination regardless of when it occurs. If we add other effects, *e.g.*, input/output or a mutable store, to the language, then the difference immediately becomes relevant.

It is easy to change our small-step rules to use right-to-left instead of left-to-right. All we need to do is change the search rules for primitive operations and function application:

$$\frac{e_2 \mapsto^1 e_2'}{e_1 \oplus e_2 \mapsto^1 e_1 \oplus e_2'}\ \text{PRIM-R} \qquad \frac{e_1 \mapsto^1 e_1'}{e_1 \oplus v_2 \mapsto^1 e_1' \oplus v_2}\ \text{PRIM-L}$$

$$\frac{e_2 \mapsto^1 e_2'}{e_1\ e_2 \mapsto^1 e_1\ e_2'}\ \text{APP-R} \qquad \frac{e_1 \mapsto^1 e_1'}{e_1\ v_2 \mapsto^1 e_1'\ v_2}\ \text{APP-L}$$

4.2 Evaluation order: call-by-value vs. call-by-name

A different choice to make—and one that has far more profound implications than left-to-right vs. right-to-left—is that between what we call *call-by-value* (CBV) and *call-by-name* (CBN).

So far, all our semantics were CBV, indicated by the fact that in a function application $e_1\ e_2$ our rules insisted in fully evaluating the argument e_2 to a value v_2 before substituting that into the body of the λ-term that is the value of e_1. Under the *call-by-name* evaluation strategy, e_2 is not evaluated and substituted as-is into the body of the function.

This can lead to needless work being avoided, because some argument's value might not actually be needed by the function. In the extreme case where the argument in question would not have terminated, this can lead to being able to run a program to a successful conclusion when CBV would not be able to do so.

A simple example of this situation is that of the constant function. Let $\Omega = (\textbf{fun}\ f(x : \textbf{int}) : \textbf{int}\ \textbf{is}\ f\ x)\ 0$ be a term whose evaluation loops forever. The following program would not terminate under CBV but returns 0 under CBN:

$$(\lambda x : \textbf{int}.0)\Omega$$

While this may be obvious here, given that the body of the function does not even mention its argument, there are other situations that exhibit similar behavior. Example:

$$(\lambda x : \textbf{int}.\lambda y : \textbf{int}.\textbf{if}\ x = 1\ \textbf{then}\ 0\ \textbf{else}\ y)\ 1\ \Omega$$

On the flip side, while CBN can save some unnecessary work, it can also lead to the duplication of *necessary* work. Let \$\$\$ be some "very expensive" expression, *i.e.*, an expression whose evaluation consumes a lot of resources (time- or space-wise). Then the following expression is likely more efficiently evaluated under CBV than it is under CBN, because CBN would have to evaluate \$\$\$ twice while CBV does it only once:

$$(\lambda x : \mathbf{int}.x + x)\ \$\$\$$$

CBN can evaluate strictly more expressions than CBV. Moreover, in a pure setting (except for non-termination) such as MinML, whenever they both succeed, their respective outcomes are *compatible*. For results of base type this means that the results will coincide:

Lemma 4.1
For pure MinML, if $\emptyset \vdash e : \mathbf{int}$ *and* $e \Downarrow_{\mathrm{CBV}} n$*, then* $e \Downarrow_{\mathrm{CBN}} n$*. An analogous statement holds for other base types such as* **bool**.

The inverse of this lemma is false (as shown above). Moreover, for higher-order types (*i.e.*, functions) the lemma does not hold in this strong form. A simple example is the following:

$$(\lambda x : \mathbf{int}.\lambda y : \mathbf{int}.x + y)\ (1 + 2)$$

Under CBV this evaluates to $\lambda y : \mathbf{int}.3 + y$ while under CBN it reduces to $\lambda y : \mathbf{int}.(1 + 2) + y$. These two terms are obviously not the same—although we certainly think of them as being equivalent in a certain, at this point mostly intuitive sense. However, a simple modification shows that the results do not even have to be equivalent (in that same intuitive sense):

$$\lambda x : \mathbf{int}.((\lambda y : \mathbf{int}.x)\ \Omega)$$

Here both CBV and CBN evaluate the expression to syntactically the *same* value (because the expression already is a value), but that value does not behave the same way when used. Under CBN, whenever we apply it to some n we get n back. Under CBV, however, any such application will fail to terminate.

Formal semantics: A substitution-based semantics can be modified very easily to express CBN instead of CBV. The only requirement is that we extend the notion of substitution to expressions, meaning that we can write $\{e'/x\}e$ for the substitution of the *closed* expression e' for x into e. For this, the definition of substitution does not actually change.

In the big-step semantics, the CBN rule for application is:

$$\frac{e_1 \Downarrow \lambda x : \tau.e \qquad \{e_2/x\}e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \textsc{app-cbn}$$

In the small-step semantics, we eliminate the structural rule for the argument expression e_2 and modify the computation rule for application accordingly. Thus, there are now two rules that deal with application:

$$\frac{e_1 \mapsto^1 e_1'}{e_1\ e_2 \mapsto^1 e_1'\ e_2}\ \textsc{app-ss-l} \qquad\qquad \frac{}{(\lambda x : \tau.e)\ e_2 \mapsto^1 \{e_2/x\}e}\ \textsc{app-ss-cbn}$$

These modifications of the small-step semantics carry over fairly directly to a modified version of the C machine.

While substitution-based semantics for CBN actually look *simpler* than those for CBV, the opposite is the case when we go to an environment-based version. The problem is, of course, the substitution of an expression for a variable. Previously, all such substitutions where done for values only. Since environments represent substitutions, we must now be able to bind a variable to an unevaluated expression. And since these expressions can have free variables and themselves have to be interpreted relative to some environment, this naturally leads to a need for some form of *expression closure*. An expression closure represents a computation that is "suspended in mid-flight," so to speak.

Thus, the setup is as follows:

machine values:	V	$::=$	$n \mid \mathbf{true} \mid \mathbf{false} \mid$	values of base type
			$\langle \lambda x : \tau . e, E \rangle$	function closures
computations:	C	$::=$	$\langle e, E \rangle$	expression closures
environments:	E	$::=$	$\mathrm{Var} \mapsto^{\mathrm{fin}} C$	

Big-step evaluation rules now have to incorporate two changes:

1. When performing a function call, the formal parameter gets bound to the closure of the argument expression.
2. When performing a variable lookup, the result will be an expression closure. At this point the computation expressed by the closure must be resumed.

Here are the changed rules:

$$\frac{E(x) = \langle e, E \rangle \qquad (E, e) \Downarrow V}{(E, x) \Downarrow V}$$

$$\frac{(E, e_1) \Downarrow \langle \lambda x : \tau_2 . e, E' \rangle \qquad (E'[x \mapsto \langle e_2, E \rangle], e) \Downarrow V}{(E, e_1\ e_2) \Downarrow V}$$

If we want to accommodate recursive functions as in MinML, the application rule becomes:

$$\frac{\begin{array}{c}(E, e_1) \Downarrow V_1 \qquad V_1 = \langle \mathbf{fun}\ f(x : \tau_2) : \tau\ \mathbf{is}\ e, E' \rangle \\ (E'[f \mapsto V_1, x \mapsto \langle e_2, E \rangle], e) \Downarrow V\end{array}}{(E, e_1\ e_2) \Downarrow V}$$

Notice that we take advantage of a fortuitous "coincidence" here (which, of course, is not really a coincidence since we set it up that way!), namely that a function closure can also be seen as an expression closure, meaning that we can bind V_1 to f.

4.3 Continuation-Passing Style (CPS)

As we have seen, in a C machine state (k, e) (or an E machine state (K, E, e)) the stack k (or K) represents the "rest of the computation" after e has been successfully evaluated. Suppose e has type τ. We can think of this "rest" (*a.k.a.*, the *continuation*) as a function from τ to some abstract type **ans** of "final answers." (For this we add **ans** to our type language. Type **ans** is not inhabited, *i.e.*, there are no introduction or eliminations forms for it in the language.)

In this section we will see that, in fact, we can rewrite any program in such a way that not only do we think of the continuation as a function—we actually implement it that way. This means that all of the functions of the original program will acquire an additional continuation argument. Since this change has to be reflected in the types of these functions, we start by defining the following *type translation*.

First, we use the following type abbreviations:

$$\begin{array}{rcll} \tau\ \mathbf{cont} & = & \tau \to \mathbf{ans} & \text{continuation expecting } \tau \\ \tau\ \mathbf{comp} & = & \tau\ \mathbf{cont} \to \mathbf{ans} & \text{computation producing } \tau \end{array}$$

Now we are ready to define $\mathrm{CPS}(\tau)$ by induction on τ:

$$\begin{array}{rcl} \mathrm{CPS}(\mathbf{nat}) & = & \mathbf{nat} \\ \mathrm{CPS}(\mathbf{bool}) & = & \mathbf{bool} \\ \mathrm{CPS}(\tau_1 \to \tau_2) & = & \mathrm{CPS}(\tau_1) \to \mathrm{CPS}(\tau_2)\ \mathbf{comp} \end{array}$$

The idea behind the CPS transformation is that any value v of type τ turns into a value $\hat{v}$ of type $\mathrm{CPS}(\tau)$, and every expression e of type τ turns into a *value* $\hat{e}$ of type $\mathrm{CPS}(\tau)$ **comp**. Notice that both values and expressions alike are turned into values. However, there is a difference:

value: Syntactic values represent themselves, they do not describe a computation, at least not immediately. (Function values do describe computations, but these computations are still "frozen" and wait for an application to "unfreeze" them.) Thus, their CPS representations are values that do not need a continuation to "run."

expression: Expressions are evaluated by "running" them. To "run" an expression, we need to know what is supposed to happen afterwards. Therefore, the CPS representation of an expression e is a function that awaits—as its argument—the continuation that receives the value of e.

We can specify the translation by two judgments, one for syntactic values and one for expressions:

$$\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{v} : \hat{\tau}$$
$$\Gamma \vdash e : \tau \rightsquigarrow \hat{e} : \hat{\tau}\ \mathbf{comp}$$

Notice that by the way the rules for these judgments are set up, it turns out that $\hat{\tau} = \mathrm{CPS}(\tau)$.

The left-hand sides (to the left of $\rightsquigarrow$) are identical to the typing judgements that we have discussed earlier. Let us first give the rules for the Simply Typed λ-calculus (without recursion).

Here are the rules for value translation judgements $\boxed{\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{v} : \hat{\tau}}$:

$$\frac{}{\Gamma \vdash_{\mathrm{v}} n : \mathbf{nat} \rightsquigarrow n : \mathbf{nat}} \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{true} : \mathbf{bool} \rightsquigarrow \mathbf{true} : \mathbf{bool}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{false} : \mathbf{bool} \rightsquigarrow \mathbf{false} : \mathbf{bool}}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow \hat{e} : \hat{\tau}_2 \ \mathbf{comp} \qquad \hat{\tau}_1 = \mathrm{CPS}(\tau_1)}{\Gamma \vdash_{\mathrm{v}} \lambda x : \tau_1 . e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \hat{\tau}_1 . \hat{e} : \hat{\tau}_1 \to \hat{\tau}_2 \ \mathbf{comp}}$$

And the rules for expression translation judgments $\boxed{\Gamma \vdash e : \tau \rightsquigarrow \hat{e} : \hat{\tau} \ \mathbf{comp}}$ are:

$$\frac{\Gamma(x) = \tau \qquad \hat{\tau} = \mathrm{CPS}(\tau)}{\Gamma \vdash x : \tau \rightsquigarrow \lambda k : \hat{\tau} \ \mathbf{cont}.k \ x : \hat{\tau} \ \mathbf{comp}}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{v} : \hat{\tau}}{\Gamma \vdash v : \tau \rightsquigarrow \lambda k : \hat{\tau} \ \mathbf{cont}.k \ \hat{v} : \hat{\tau} \ \mathbf{comp}}$$

$$\frac{\Gamma \vdash e : \mathbf{nat} \rightsquigarrow \hat{e} : \mathbf{nat\ comp}}{\Gamma \vdash \mathbf{pred}(e) : \mathbf{nat} \rightsquigarrow \lambda k : \mathbf{nat\ cont}.\hat{e} \ \lambda x : \mathbf{nat}.k \ (\mathbf{pred}(x)) : \mathbf{nat\ comp}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{bool} \rightsquigarrow \hat{e}_1 : \mathbf{bool\ comp} \\ \Gamma \vdash e_2 : \tau \rightsquigarrow \hat{e}_2 : \hat{\tau} \ \mathbf{comp} \qquad \Gamma \vdash e_3 : \tau \rightsquigarrow \hat{e}_3 : \hat{\tau} \ \mathbf{comp}\end{array}}{\begin{array}{rcl}\Gamma & \vdash & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \rightsquigarrow \\ & & \lambda k : \hat{\tau}\ \mathbf{cont}.\hat{e}_1\ \lambda x : \mathbf{bool}.\mathbf{if}\ x\ \mathbf{then}\ \hat{e}_2\ k\ \mathbf{else}\ \hat{e}_3\ k : \hat{\tau}\ \mathbf{comp}\end{array}}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow \hat{e}_1 : (\hat{\tau}_2 \to \hat{\tau}\ \mathbf{comp})\ \mathbf{comp} \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2 : \hat{\tau}_2\ \mathbf{comp}}{\begin{array}{rcl}\Gamma & \vdash & e_1\ e_2 : \tau \rightsquigarrow \\ & & \lambda k : \hat{\tau}\ \mathbf{cont}.\hat{e}_1\ \lambda x_1 : \hat{\tau}_2 \to \hat{\tau}\ \mathbf{comp}.\hat{e}_2\ \lambda x_2 : \hat{\tau}_2 . x_1\ x_2\ k : \hat{\tau}\ \mathbf{comp}\end{array}}$$

The rules for **succ** and **iszero** are analogous to that for **pred**.

The CPS language: It is relatively easy to check that the translation relation is single-valued: in $\Gamma \vdash e : \tau \rightsquigarrow e' : \tau'$, the "inputs" Γ and e uniquely determine the "outputs" τ, e', and τ'. Moreover, e' always lies in a certain sub-language of the whole language, because translation terms maintain some syntactic invariants. In particular, in any application form $e_1\ e_2$, the argument e_2 is either a value or a variable or a *simple* computation of the form $\mathbf{pred}(v)$, $\mathbf{succ}(v)$, or $\mathbf{iszero}(v)$. If we add binary primitive operators as in MinML, we would also see $v_1 \oplus v_2$ as arguments.

We can strengthen the invariant by replacing simple computation expressions with corresponding binding forms. For example, $\mathbf{pred}(v)$ becomes $\mathbf{let}\ x = \mathbf{pred}(v)\ \mathbf{in}\ e$. In addition, we will also classify variables as syntactic values. With such a target language it is then possible to arrange for all arguments in application forms to be syntactic values.

We will now re-formulate the translation judgments and their rules in such a way that translation terms $\hat{v}$ and $\hat{e}$ are drawn from the following *target* language (as opposed to also be programs of the source language):

types:	τ	$::=$	$\mathbf{nat} \mid$	nat. numbers
			$\mathbf{bool} \mid$	booleans
			$\mathbf{ans} \mid$	answers
			$\tau \to \tau$	functions
syntactic values:	$\hat{v}$	$::=$	$x \mid$	variables
			$n \mid$	numbers (intro. **nat**)
			$\mathbf{false} \mid \mathbf{true} \mid$	booleans (intro. **bool**)
			$\lambda x : \tau.\hat{e}$	λ-abstraction (intro. $\tau_1 \to \tau_2$)
simple computations:	$\hat{s}$	$::=$	$\mathbf{pred}(\hat{v}) \mid \mathbf{succ}(\hat{v}) \mid \mathbf{iszero}(\hat{v})$	
expressions:	$\hat{e}$	$::=$	$\hat{v} \mid$	values
			$\mathbf{let}\ x = \hat{s}\ \mathbf{in}\ \hat{e} \mid$	simple computation with binding
			$\mathbf{if}\ \hat{v}\ \mathbf{then}\ \hat{e}\ \mathbf{else}\ \hat{e} \mid$	conditional (elim. **bool**)
			$\hat{e}\ \hat{v}$	appliation (elim. $\tau_1 \to \tau_2$)

Here are the rules for the new translation judgments. This time, for improved readability we omit the types of translation terms. Also, we will elide most of the type annotations on λ-bound variables, as these types can be inferred from context.

We start with the translation values[8] $\boxed{\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{v}\ (:\hat{\tau})}$:

$$\frac{}{\Gamma \vdash_{\mathrm{v}} n : \mathbf{nat} \rightsquigarrow n} \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{true} : \mathbf{bool} \rightsquigarrow \mathbf{true}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} \mathbf{false} : \mathbf{bool} \rightsquigarrow \mathbf{false}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{v}} x : \tau \rightsquigarrow x}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow \hat{e} : \hat{\tau}_2\ \mathbf{comp} \qquad \hat{\tau}_1 = \mathrm{CPS}(\tau_1)}{\Gamma \vdash_{\mathrm{v}} \lambda x : \tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \hat{\tau}_1.\hat{e}}$$

The rules for expressions judgments $\boxed{\Gamma \vdash e : \tau \rightsquigarrow \hat{e}\ (:\hat{\tau}\ \mathbf{comp})}$ are then:

[8] We also include the translation of variables here, since this avoids those two nearly identical-looking rules for variables and values in the expression translation.

$$\frac{\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{v}}{\Gamma \vdash v : \tau \rightsquigarrow \lambda k.k\ \hat{v}} \qquad \frac{\Gamma \vdash e : \mathbf{nat} \rightsquigarrow \hat{e}}{\Gamma \vdash \mathbf{pred}(e) : \mathbf{nat} \rightsquigarrow \lambda k.\hat{e}\ \lambda x.\ \mathbf{let}\ y = \mathbf{pred}(x)\ \mathbf{in}\ k\ y}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \rightsquigarrow \hat{e}_1 \qquad \Gamma \vdash e_2 : \tau \rightsquigarrow \hat{e}_2 \qquad \Gamma \vdash e_3 : \tau \rightsquigarrow \hat{e}_3}{\begin{array}{rcl}\Gamma & \vdash & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \rightsquigarrow \\ & & \lambda k.\hat{e}_1\ \lambda x.\mathbf{if}\ x\ \mathbf{then}\ \hat{e}_2\ k\ \mathbf{else}\ \hat{e}_3\ k\end{array}}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow \hat{e}_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2}{\Gamma \vdash e_1\ e_2 : \tau \rightsquigarrow \lambda k.\hat{e}_1\ \lambda x_1.\hat{e}_2\ \lambda x_2.x_1\ x_2\ k}$$

The rules for **succ** and **iszero** are analogous to that for **pred**.

Recursive functions: Adding recursion in the style of MinML is easy: we simply replace λ-terms $\lambda x : \tau.e$ with their recursive cousins $\mathbf{fun}\ f(x : \tau_1) : \tau_2\ \mathbf{is}\ e$ in both the source and the target language.[9]

The translation rule for functions is then:

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow \hat{e} \qquad \hat{\tau}_1 = \mathrm{CPS}(\tau_1) \qquad \hat{\tau}_2 = \mathrm{CPS}(\tau_2)}{\begin{array}{rcl}\Gamma & \vdash & \mathbf{fun}\ f(x : \tau_1) : \tau_2\ \mathbf{is}\ e : \tau_1 \to \tau_2 \rightsquigarrow \\ & & \mathbf{fun}\ f(x : \hat{\tau}_1) : \hat{\tau}_2\ \mathbf{comp}\ \mathbf{is}\ \hat{e}\end{array}}$$

Evaluation order: Recall that the difference between left-to-right CBV, right-to-left CBV, and CBN are explained by looking at when the argument e_2 in an application $e_1\ e_2$ gets evaluated. In left-to-right CBV it is evaluated just after e_1, in right-to-left CBV it is evaluated just before e_1, and in CBN it is not evaluated at all until the body of the function demands its value.

But notice that this difference completely vanishes in our CPS sub-language! After all, e_2 is already required to be a value, so the timing of its evaluation does not matter. We say that CPS is evaluation-order-independent. Programs written in CPS have made the original evaluation order explicit.

So, when converting from direct style to continuation passing style, what happened to the difference between evaluation orders? The answer is that the process of converting the program from one style to the other chooses which order is being used. The rules for CPS conversion that we have seen implement left-to-right CBV. A right-to-left CBV version can be constructed quite trivially by letting $\hat{e}_2$ run before $\hat{e}_1$ in the conclusion of the rule for application.

CBN version of CPS conversion: To implement CBN via CPS conversion, we have to change the type translation as well. The reason for this is that under CBN the argument of a function is not yet evaluated, so it represents a *computation* and not a *value*. Thus, we have:

$$\mathrm{CPS_N}(\mathbf{nat}) \quad = \quad \mathbf{nat}$$

[9] We will still write $\lambda x : \tau.e$ as a shorthand for $\mathbf{fun}\ f(x : \tau) : \tau'\ \mathbf{is}\ e$ where f does not occur free in e where τ' is chosen appropriately for the expression to type-check as $\tau \to \tau'$.

$$\begin{aligned}\mathrm{CPS_N}(\mathbf{bool}) &= \mathbf{bool} \\ \mathrm{CPS_N}(\tau_1 \to \tau_2) &= \mathrm{CPS_N}(\tau_1)\ \mathbf{comp} \to \mathrm{CPS_N}(\tau_2)\ \mathbf{comp}\end{aligned}$$

The next change concerns the treatment of variables. Unlike in CBV where variables denote values, in CBN they denote computations. Thus, they are not translated using a $\vdash_{\mathrm{v}}$ judgment. Instead, we use the following rule:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow x : \mathrm{CPS_N}(\tau)\ \mathbf{comp}}$$

Given this we now have to make sure that in the rule for converting function applications $\hat{e}_2$ does not "run" but simply gets passed directly as an argument:

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow \hat{e}_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2}{\Gamma \vdash e_1\ e_2 : \tau \rightsquigarrow \lambda k.\hat{e}_1\ \lambda x_1.x_1\ \hat{e}_2\ k}$$

Finally, there is a problem concerning recursive functions. Recall the small-step evaluation rule for recursive function application, which states:

$$\frac{v_1 = \mathbf{fun}\ f(x : \tau_1) : \tau_2\ \mathbf{is}\ e}{v_1\ v_2 \mapsto^1 \{v_1/f\}\{v_2/x\}e}$$

In the CBN setting, v_2 denotes a computation, but v_1 does not! Therefore, in order to maintain the invariant that all variables are bound to computations, we have to manipulate the binding of f within the body of the function a bit. The trick is to rebind f to $\lambda k.k\ f$, which can be done as follows:

$$\frac{\begin{array}{c}\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow \hat{e} : \hat{\tau}_2\ \mathbf{comp} \\ \hat{\tau}_1 = \mathrm{CPS_N}(\tau_1) \qquad \hat{\tau}_2 = \mathrm{CPS_N}(\tau_2)\end{array}}{\begin{array}{ll}\Gamma \vdash_{\mathrm{v}} & \mathbf{fun}\ f(x : \tau_1) : \tau_2\ \mathbf{is}\ .e : \tau_1 \to \tau_2 \rightsquigarrow \\ & \mathbf{fun}\ f'(x : \hat{\tau}_1\ \mathbf{comp}) : \hat{\tau}_2\ \mathbf{comp}.(\lambda f.\hat{e})\ (\lambda k.k\ f')\end{array}}$$

Here, in the translation, f' has type $\hat{\tau}_1\ \mathbf{comp} \to \hat{\tau}_2\ \mathbf{comp}$, while the type of f is the required $(\hat{\tau}_1\ \mathbf{comp} \to \hat{\tau}_2\ \mathbf{comp})\ \mathbf{comp}$ (which is the same as $\mathrm{CPS_N}(\tau_1 \to \tau_2)\ \mathbf{comp}$). The continuation variable k has type $(\hat{\tau}_1\ \mathbf{comp} \to \hat{\tau}_2\ \mathbf{comp})\ \mathbf{cont}$.

5 Mutable Store

So far our language did not have *state*—it was pure (up to non-termination). State can be added in form of a global mutable store, *a.k.a.* "memory" M, which is a finite mapping from locations l to closed values. New ("fresh") locations

are generated by extending the current store so that it maps some location that previously had not been mapped.

One of the simplest way of modelling this kind of language is to use a substitution-based semantics and add locations to the syntax of values.

$$v \quad ::= \quad \ldots \mid l$$

However, the idea is that "source" programs do not actually contain any locations. Locations get generated during evaluation. They become part of the current expression by some sub-expression (in particular, a sub-expression that performs memory allocation) getting evaluated to a new location. Like other values, locations can then make their way into other parts of the current expression via subsequent substitutions *etc.*.

On the expression side, there are three additions. First, **ref** e allocates a fresh location l and initializes $M(l)$ to the value of e. Second, $!e$ evaluates e to a location l and returns $M(l)$. And third, $e_1 := e_2$ evaluates e_1 to a location l, e_2 to a value v, and then sets $M(l)$ to be equal to v:

types:	τ	$::=$	$\mathbf{nat} \mid$	nat. numbers
			$\mathbf{bool} \mid$	booleans
			$\tau \to \tau \mid$	functions
			τ **ref**	reference
values:	v	$::=$	$n \mid$	numbers (intro. **nat**)
			$\mathbf{false} \mid \mathbf{true} \mid$	booleans (intro. **bool**)
			$\lambda x : \tau . e \mid$	λ-abstraction (intro. $\tau_1 \to \tau_2$)
			l	locations
expressions:	e	$::=$	$x \mid$	variables
			$v \mid$	values
			$\mathbf{pred}(e) \mid \mathbf{succ}(e) \mid \mathbf{iszero}(e) \mid$	arithmetic (elim. **nat**)
			$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid$	conditional (elim. **bool**)
			$e\ e \mid$	appliation (elim. $\tau_1 \to \tau_2$)
			$\mathbf{ref}\ e \mid$	reference alloc+init
			$!e \mid$	dereference
			$e := e$	assignment to reference

5.1 Typing

Since we added locations to the syntax of the language, our static semantics now needs an additional typing environment, *i.e.*, a mapping Λ from locations to types. However, recall that we said that "source" expressions do not contain locations, so Λ can be taken to be empty for those. However, for the proof of safety, we need Λ because the statements of the progress and preservation lemmas consider arbitrary intermediate states and not only "source" expressions.

The new typing judgment has the form $\Lambda; \Gamma \vdash e : \tau$. Most rules leave Λ completely alone. The only rule to mention it in a non-trivial way is the one for locations:

$$\frac{\Lambda(l) = \tau}{\Lambda; \Gamma \vdash l : \tau}$$

Since we added three new expression forms, we need three additional rules:

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \mathbf{ref}\ e : \tau\ \mathbf{ref}} \qquad \frac{\Lambda; \Gamma \vdash e : \tau\ \mathbf{ref}}{\Lambda; \Gamma \vdash\ !e : \tau} \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau\ \mathbf{ref} \qquad \Lambda; \Gamma \vdash e_2 : \tau}{\Lambda; \Gamma \vdash e_1 := e_2 : \tau}$$

The third rule (arbitrarily) assumes that $e_1 := e_2$ returns the value of e_2.[10]

A simple fact that comes in handy during the proof of safety (see below) and which is trivial to prove by induction is expressed by the following lemma:

Lemma 5.1
If $\Lambda; \Gamma \vdash e : \tau$ *and* $\Lambda' \supseteq \Lambda$, *then* $\Lambda'; \Gamma \vdash e : \tau$.

That is, adding more bindings to an existing store typing does not invalidate any typing judgments.

5.2 Dynamic semantics

A substitution-based small-step semantics for the augmented language consists of rules for deriving judgments of the form $(M, e) \mapsto^1 (M', e')$, which should be read: "Expression e in memory M steps to e' and the memory changes to M'."

First, here are all the original rules, augmented with memory M. Also shown are the *structural* ("search") rules for **ref**, !, and :=. All of these rules simply carry M through without looking at its contents and without changing it.

First, the computation rules:

$$\frac{n' = n - 1}{(M, \mathbf{pred}(n)) \mapsto^1 (M, n')} \qquad \frac{n' = n + 1}{(M, \mathbf{succ}(n)) \mapsto^1 (M, n')}$$

$$\frac{}{(M, \mathbf{iszero}(0)) \mapsto^1 (M, \mathbf{true})} \qquad \frac{n \neq 0}{(M, \mathbf{iszero}(n)) \mapsto^1 (M, \mathbf{false})}$$

$$\frac{}{(M, \mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2) \mapsto^1 (M, e_1)}$$

$$\frac{}{(M, \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2) \mapsto^1 (M, e_2)}$$

$$\frac{}{(M, (\lambda x : \tau . e)\ v) \mapsto^1 (M, \{v/x\}e)}$$

[10] This is analogous to how the assignment operator = in the C programming language works. In Standard ML, assignment to references returns the unit value `()` and has type **unit**.

Here are the original structural rules, augmented with memory:

$$\frac{(M, e) \mapsto^1 (M', e')}{(M, \mathbf{pred}(e)) \mapsto^1 (M', \mathbf{pred}(e'))} \qquad \frac{(M, e) \mapsto^1 (M', e')}{(M, \mathbf{succ}(e)) \mapsto^1 (M', \mathbf{succ}(e'))}$$

$$\frac{(M, e) \mapsto^1 (M', e')}{(M, \mathbf{iszero}(e)) \mapsto^1 (M', \mathbf{iszero}(e'))}$$

$$\frac{(M, e_1) \mapsto^1 (M', e_1')}{(M, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \mapsto^1 (M', \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)}$$

$$\frac{(M, e_1) \mapsto^1 (M', e_1')}{(M, e_1\ e_2) \mapsto^1 (M', e_1'\ e_2)} \qquad \frac{(M, e_2) \mapsto^1 (M', e_2')}{(M, v_1\ e_2) \mapsto^1 (M', v_1\ e_2')}$$

The structural rules for the new expression forms are:

$$\frac{(M, e) \mapsto^1 (M', e')}{(M, \mathbf{ref}\ e) \mapsto^1 (M', \mathbf{ref}\ e')} \qquad \frac{(M, e) \mapsto^1 (M', e')}{(M, !e) \mapsto^1 (M', !e')}$$

$$\frac{(M, e_1) \mapsto^1 (M', e_1')}{(M, e_1 := e_2) \mapsto^1 (M', e_1' := e_2)} \qquad \frac{(M, e_2) \mapsto^1 (M', e_2')}{(M, v_1 := e_2) \mapsto^1 (M', v_1 := e_2')}$$

Finally, these are the rules that actually refer to the contents of M or modify M itself:

$$\frac{l \notin \mathrm{dom}(M)}{(M, \mathbf{ref}\ v) \mapsto^1 (M[l \mapsto v], l)} \qquad \frac{l \in \mathrm{dom}(M) \qquad v = M(l)}{(M, !\ l) \mapsto^1 (M, v)}$$

$$\frac{l \in \mathrm{dom}(M)}{(M, l := v) \mapsto^1 (M[l \mapsto v], v)}$$

From this semantics, it is not difficult to change and extend our C machine to accommodate the language extension. The main modification is to add memory M to each machine state, leaving us with *expression states* of the form (k, M, e) (meaning "expression e is about the be evaluated in memory M and pending computations waiting for e's result are remembered in stack k), and *value states* of the form (v, k, M) (meaning "a value v has been computed and is about to be passed to the pending work remembered in k; the current memory is M). To work out the (straightforward) details is left as an exercise.

5.3 Safety

Safety for a step-relation such as $\mapsto^1$ means that from some start state s_0 we can iterate $\mapsto^1$ as in $s_0 \mapsto^1 s_1 \mapsto^1 \cdots \mapsto^1 s_i \mapsto^1 \cdots$ either indefinitely or until

we reach a "desirable" final state s_n. So far, our start state was the original source expression and the final state was its value.

To prove safety, we identify a property $P(s)$ that

1. holds for s_0 (*i.e.*, $P(s_0)$),
2. is preserved by $\mapsto^1$ (*i.e.*, if $P(s)$ and $s \mapsto^1 s'$, then $P(s')$), and
3. guarantees that the state in question is either one of the desirable final states or a step can be taken (*i.e.*, if $P(s)$ then either s is final or $\exists s'.s \mapsto^1 s'$).

For our simple arithmetic language with **let**, the states were expressions e, the final states were the values (*i.e.*, numbers) n, and the property P was written e **ok**. Similarly, in STLC and MinML, the states were expressions, final states were values, and the property in question was $\exists\tau.\emptyset \vdash e : \tau$.

Due to the addition of mutable state, the new step relation $\mapsto^1$ is now between pairs (M, e) of memory M and expressions e. Notice that intermediate states (any state except the start state) may contain locations l that lie within the domain of M. To prove safety, we have to identify the property P that holds for pairs (M, e) and that is preserved by the step relation.

For this, we first define what it means for a memory M to have type Λ, written $\vdash M : \Lambda$. The definition for this relation is the following:

Definition 5.2
A memory M conforms to a store typing Λ (written $\vdash M : \Lambda$) if and only if the domains of M and Λ coincide ($\mathrm{dom}(M) = \mathrm{dom}(\Lambda)$*) and Λ correctly describes the type of every value stored in M (*$\forall l \in \mathrm{dom}(\Lambda).\Lambda; \emptyset \vdash M(l) : \Lambda(l)$*).*

Notice that the typing judgment for $M(l)$ is with respect to the entire Λ. This means that l can contain a value that—directly or indirectly—mentions l itself, thus giving rise to the possibility of cyclic structures. Indeed, assignment to memory locations is capable of creating these.

With this preparation, the property that is preserved by $\mapsto^1$, which we will tentatively write $\vdash (M, e)$ **ok**, is the following:

Definition 5.3
The configuration (M, e) is well-formed, written $\vdash (M, e)$ **ok***, if both $\vdash M : \Lambda$ and $\Lambda; \emptyset \vdash e : \tau$ for some Λ and τ.*

With this in place, we can write down the statement of the safety lemma:

Lemma 5.4
If $\vdash (M, e)$ **ok***, then either e is a value or there exist M' and e' such that $(M, e) \mapsto^1 (M', e')$ and $\vdash (M', e')$* **ok***.*

However, it turns out that this statement is not strong enough for use in its own inductive proof. The problem is that while τ stays the same throughout the entire evaluation sequence, Λ changes. And since Λ changes, we have to be

able to relate the Λ that witnesses the well-formedness of one configuration to the Λ for the next. A good start is to be more concrete than just saying "**ok**." Instead, our revised statement of well-formedness mentions the particular Λ and τ that are its witnesses:

Definition 5.5
The configuration (M, e) is well-formed, written $\vdash (M, e) : \Lambda, \tau$, if both $\vdash M : \Lambda$ and $\Lambda; \emptyset \vdash e : \tau$.

The strengthened form of the safety lemma is then:

Lemma 5.6
If $\vdash (M, e) : \Lambda, \tau$, then either e is a value or there exist M', e', and Λ' such that $\Lambda' \supseteq \Lambda$ and $(M, e) \mapsto^1 (M', e')$ and $\vdash (M', e') : \Lambda', \tau$.

This formulation (and the proof) of the safety property inherently relies on the fact that our language has only *weak update*: an assignment to a location never changes its type. Therefore, the store typing Λ is extended only conservatively, by extension.

The proof for this lemma follows the standard pattern for proving *progress* and *preservation*.

6 The polymorphic λ-calculus (System F)

6.1 Syntax

types:	τ	$::=$	$\alpha \mid$	type variables
			$\tau \to \tau \mid$	function types
			$\forall\alpha.\tau \mid$	polymorphic types
			b	some base type(s)
values:	v	$::=$	$x \mid$	variables
			$\lambda x : \tau.e \mid$	(ordinary) abstractions
			$\Lambda\alpha.e \mid$	type abstractions
			c_τ	constants (of type τ)
expressions:	e	$::=$	$v \mid$	values
			$e\ e$	(ordinary) applications
			$e[\tau]$	type applications
type variable environments:	Δ	$\subseteq$	TyVar	sets of type variables
(term) variable environments:	Γ	$\in$	$\mathrm{Var} \mapsto^{\mathrm{fin}} \tau$	finite mappings from variables to types

(The types τ of constants c_τ are assumed not to contain any type variables. This implies that they are well-formed, *i.e.*, $\emptyset \vdash \tau$ **ok** in the sense defined below.)

6.2 Static semantics

Well-formedness of types

Judgments of the form $\boxed{\Delta \vdash \tau\ \mathbf{ok}}$:

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathbf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \mathbf{ok} \qquad \Delta \vdash \tau_2\ \mathbf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \mathbf{ok}} \qquad \frac{\Delta \cup \{\alpha\} \vdash \tau\ \mathbf{ok} \qquad \alpha \notin \Delta}{\Delta \vdash \forall \alpha.\tau\ \mathbf{ok}}$$

$$\frac{}{\Delta \vdash b\ \mathbf{ok}}$$

Typing of values

Judgments of the form $\boxed{\Delta; \Gamma \vdash_{\mathrm{v}} v : \tau}$:

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_{\mathrm{v}} x : \tau} \qquad \frac{\Delta \vdash_{\mathrm{v}} \tau_1\ \mathbf{ok} \qquad \Delta; \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Delta; \Gamma \vdash_{\mathrm{v}} \lambda x : \tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{\Delta \cup \{\alpha\}; \Gamma \vdash e : \tau \qquad \alpha \notin \Delta}{\Delta; \Gamma \vdash_{\mathrm{v}} \Lambda \alpha.e : \forall \alpha.\tau} \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{v}} c_\tau : \tau}$$

Typing of expressions

Judgments of the form $\boxed{\Delta; \Gamma \vdash e : \tau}$:

$$\frac{\Delta; \Gamma \vdash_{\mathrm{v}} v : \tau}{\Delta; \Gamma \vdash v : \tau} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau' \qquad \Delta \vdash \tau\ \mathbf{ok}}{\Delta; \Gamma \vdash e[\tau] : \{\tau/\alpha\}\tau'}$$

6.3 Small-step semantics

Structural (search) rules

$$\frac{e_1 \mapsto^1 e_1'}{e_1\ e_2 \mapsto^1 e_1'\ e_2} \qquad \frac{e_2 \mapsto^1 e_2'}{v_1\ e_2 \mapsto^1 v_1\ e_2'} \qquad \frac{e \mapsto^1 e'}{e[\tau] \mapsto^1 e'[\tau]}$$

Computation rules

We omit the rules involving constants c_τ.

$$\frac{}{(\lambda x : \tau.e)v \mapsto^1 \{v/x\}e} \qquad \frac{}{(\Lambda \alpha.e)[\tau] \mapsto^1 \{\tau/\alpha\}e}$$

6.4 Church-encoding types

Boolean

$$
\begin{array}{rcl}
\textbf{bool} & \equiv & \forall\alpha.\ \alpha \to \alpha \to \alpha \\
\textbf{true} & = & \Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ x \\
\textbf{false} & = & \Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ y \\
\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & = & e_1[\tau]\ e_2\ e_3 \qquad ;\ \text{where } e_{2,3} : \tau \\
\neg\ e & = & e[\textbf{bool}]\ \textbf{false}\ \textbf{true} \\
e_1 \textbf{ andalso } e_2 & = & e_1[\textbf{bool}]\ e_2\ \textbf{false}
\end{array}
$$

Sum types

$$
\begin{array}{rcl}
\tau_1 + \tau_2 & \equiv & \forall\alpha.\ (\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha \\
\textbf{inl}(e) & = & \Lambda\alpha.\ \lambda k_1 : \tau_1 \to \alpha.\ \lambda k_2 : \tau_2 \to \alpha.\ k_1\ e \\
\textbf{inr}(e) & = & \Lambda\alpha.\ \lambda k_1 : \tau_1 \to \alpha.\ \lambda k_2 : \tau_2 \to \alpha.\ k_2\ e \\
\textbf{case } e_0 \textbf{ of } \textbf{inl}(x_1) \Rightarrow e_1 \mid \textbf{inr}(x_2) \Rightarrow e_2 & = & e_0[\tau]\ (\lambda x_1 : \tau_1.\ e_1)\ (\lambda x_2 : \tau_2.\ e_2)
\end{array}
$$

Product types

$$
\begin{array}{rcl}
\tau_1 \times \tau_2 & \equiv & \forall\alpha.\ (\tau_1 \to \tau_2 \to \alpha) \to \alpha \\
(e_1, e_2) & = & \Lambda\alpha.\ \lambda k : \tau_1 \to \tau_2 \to \alpha.\ k\ e_1\ e_2 \\
\textbf{case } e \textbf{ of } (x_1, x_2) \Rightarrow e' & = & e[\tau]\ (\lambda x_1 : \tau_1 . \lambda x_2 : \tau_2 . e') \\
\#1(e) & = & e[\tau_1]\ (\lambda x_1 : \tau_1.\ \lambda x_2 : \tau_2.\ x_1) \\
\#2(e) & = & e[\tau_2]\ (\lambda x_1 : \tau_1.\ \lambda x_2 : \tau_2.\ x_2)
\end{array}
$$

Naturals (*an inductive type*)

$$
\begin{array}{rcl}
\textbf{nat} & \equiv & \forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha \\
\textbf{zero} & = & \Lambda\alpha.\ \lambda s : \alpha \to \alpha.\ \lambda z : \alpha.\ z \\
\textbf{succ} & = & \lambda n : \textbf{nat}.\ \Lambda\alpha.\ \lambda s : \alpha \to \alpha.\ \lambda z : \alpha.\ s\ (n[\alpha]\ s\ z) \\
\textbf{iszero} & = & \lambda n : \textbf{nat}.\ n[\textbf{bool}]\ (\lambda x : \textbf{bool}.\textbf{false})\ \textbf{true} \\
\textbf{add}(m, n) = m + n & = & \Lambda\alpha.\ \lambda s : \alpha \to \alpha.\ \lambda z : \alpha.\ m[\alpha]\ s\ (n[\alpha]\ s\ z) \\
\textbf{mul}(m, n) = m \cdot n & = & \Lambda\alpha.\ \lambda s : \alpha \to \alpha.\ \lambda z : \alpha.\ m[\alpha]\ (n[\alpha]\ s)\ z \\
\textbf{pow}(n, m) = n^m & = & \Lambda\alpha.\ \lambda s : \alpha \to \alpha.\ \lambda z : \alpha.\ m[\alpha \to \alpha]\ (n[\alpha])\ s\ z \\
\\
psucc & = & \lambda N : \textbf{bool} \times \textbf{nat}.\ (\textbf{true}, \textbf{if } \#1(N) \textbf{ then succ}(\#2(N)) \textbf{ else zero}) \\
\textbf{pred} & = & \lambda n : \textbf{nat}.\ \#2(n[\textbf{bool} \times \textbf{nat}]\ psucc\ (\textbf{false}, \textbf{zero})) \\
\\
\textbf{sub}(m, n) = m - n & = & n[\textbf{nat}]\ \textbf{pred}\ m \qquad\qquad (\text{“natural” subtraction})
\end{array}
$$

$$\begin{aligned}
\mathbf{ge}(m,n) = m \geq n &= \mathbf{iszero}(n-m) \\
\mathbf{gt}(m,n) = m > n &= \neg(n \geq m) \\
\mathbf{eq}(m,n) = (m = n) &= m \geq n \ \mathbf{andalso}\ n \geq m
\end{aligned}$$

Another inductive type: lists

(Shown here are lists of one concrete element type τ. To get a list type *constructor* like in ML, we would have to additionally parameterize everything over the element type.)

$$\begin{aligned}
\tau\ \mathbf{list} &\equiv \forall\alpha.(\tau \to \alpha \to \alpha) \to \alpha \to \alpha \\
\mathbf{nil} &= \Lambda\alpha.\ \lambda c : \tau \to \alpha \to \alpha.\ \lambda n : \alpha.\ n \\
\mathbf{cons} &= \lambda x : \tau.\ \lambda l : \tau\ \mathbf{list}.\ \Lambda\alpha.\ \lambda c : \tau \to \alpha \to \alpha.\ \lambda n : \alpha.\ c\ x\ (l[\alpha]\ c\ n) \\
\mathbf{foldr} &= \Lambda\alpha.\lambda c : \tau \to \alpha \to \alpha.\ \lambda n : \alpha. \qquad l \text{ is its own right-fold} \\
&\quad\ \lambda l : \tau\ \mathbf{list}.\ l[\alpha]\ c\ n \\
\mathbf{foldl} &= \Lambda\alpha.\ \lambda c : \tau \to \alpha \to \alpha.\ \lambda n : \alpha. \\
&\quad\ \lambda l : \tau\ \mathbf{list}. \\
&\qquad l[\alpha \to \alpha]\ (\lambda x : \tau.\ \lambda f : \alpha \to \alpha.\ \lambda y : \alpha.\ f\ (c\ x\ y))\ (\lambda y : \alpha.\ y)\ n \\
\mathbf{append} &= \lambda l_1 : \tau\ \mathbf{list}.\ \lambda l_2 : \tau\ \mathbf{list}.\ \mathbf{foldr}[\tau\ \mathbf{list}]\ \mathbf{cons}\ l_2\ l_1 \\
\mathbf{reverse} &= \mathbf{foldl}[\tau\ \mathbf{list}]\ \mathbf{cons}\ \mathbf{nil}
\end{aligned}$$

7 Type inference

Let us go back to the simply typed λ-calculus where we no longer require that bound variables in λ-terms be annotated with their types:

types:	τ	$::=$	$\tau \to \tau \mid$	function types
			b	some base type(s)
values:	v	$::=$	$x \mid$	variables
			$\lambda x.e \mid$	abstractions
			c_τ	constants (of type τ)
expressions:	e	$::=$	$v \mid$	values
			$e\ e$	applications
(term) variable environments:	Γ	$\in$	$\mathrm{Var} \mapsto^{\mathrm{fin}} \tau$	finite mappings from variables to types

The typing rules for this language are essentially the same as those for the STLC with type annotations. The only difference is that the rules no longer define a *single-valued* relation:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{v}} x : \tau}\ \textsc{var} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash_{\mathrm{v}} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{lam} \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} c_\tau : \tau}\ \textsc{con} \qquad \frac{\Gamma \vdash_{\mathrm{v}} v : \tau}{\Gamma \vdash v : \tau}\ \textsc{val}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{app}$$

The reason for the emerging ambiguity is the fact that the rule for λ-abstractions (LAM), if read from bottom to top, is making a "guess" as to what the type τ_1 of the bound variable shall be. Indeed, a term such as $\lambda x.x$ has many possible types.

The typing rules, as stated, still make sense. They constitute what is called a *declarative* type system: They do specify the typing relation $\Gamma \vdash e : \tau$, but they are no-deterministic as they do not produce a unique typing, and they also do not directly give rise to an algorithm for type checking.

7.1 Principal types

We can modify the above type system by adding type variables to the language of types. For the time being, type variables represent types whose identity the typing rules do not care about.

$$\tau \quad ::= \quad \ldots \mid \alpha \tag{2}$$

With this minor change, the type system not only can give $\lambda x.x$ types such as $b \to b$ or $(b \to b) \to (b \to b)$, *etc.*, it now can also assign the type $\alpha \to \alpha$ for some choice of type variable α.

With this, it turns out that this small language fragment enjoys a property called *principal types*. This property states that if $\Gamma \vdash e : \tau_0$ (*i.e.*, if a term e can be typed at all), then there exists a type $\hat{\tau}$ such that

1. $\Gamma \vdash e : \hat{\tau}$, and
2. for any τ, if $\Gamma \vdash e : \tau$ then there exists a *type substitution* σ such that $\tau = \sigma(\hat{\tau})$. (In particular, $\tau_0 = \sigma_0(\hat{\tau})$ for some σ_0.)

Here, a type substitution σ is a finite map from type variables to types.

7.2 Computing the principal type

An algorithm for computing the principal type can be constructed using the following idea:

Whenever the typing rules have to "guess" a type, we make up a placeholder for that type by introducing a new, fresh type variable α. Later, when it is discovered that the type in question actually needs to be some specific τ, we record on the side that $\alpha = \tau$. Such an equation is called a *constraint*.

We can capture this idea using rules for deriving a *constraint typing relation* $\Gamma \vdash e : \tau \mid_\chi C$. Here C is a set of constraints of the form $\tau_1 = \tau_2$, and χ (which is used for bookkeeping) is a set of type variables that were introduced when guessing types.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{v}} x : \tau \mid_\emptyset \emptyset}\ \textsc{var-c} \qquad \frac{\alpha \notin \Gamma, \chi \qquad \Gamma[x \mapsto \alpha] \vdash e : \tau \mid_\chi C}{\Gamma \vdash_{\mathrm{v}} \lambda x.e : \alpha \to \tau \mid_{\chi \cup \{\alpha\}} C}\ \textsc{lam-c}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} c_\tau : \tau \mid_\emptyset \emptyset}\ \textsc{con-c} \qquad \frac{\Gamma \vdash_{\mathrm{v}} v : \tau \mid_\chi C}{\Gamma \vdash v : \tau \mid_\chi C}\ \textsc{val-c}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid_{\chi_2} C_2 \\ \chi_1 \cap \chi_2 = \chi_1 \cap \mathrm{FTV}(\tau_2) = \mathrm{FTV}(\tau_1) \cap \chi_2 = \emptyset \\ \alpha \notin \chi_1, \chi_2, \tau_1, \tau_2, C_1, C_2, \Gamma \qquad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}\end{array}}{\Gamma \vdash e_1\ e_2 : \alpha \mid_{\chi_1 \cup \chi_2 \cup \{\alpha\}} C'}\ \textsc{app-c}$$

These rules are single-valued again (up to the choice of type variables during "guessing"). In fact, these rules never fail! They produce a constraint typing for an arbitrary Γ and an arbitrary e—as long as all free variables of e are accounted-for in Γ. However, this does not mean that all expressions suddenly type-check. What happened is that type errors now become manifest as contradictory sets of constraints.

Constraint solving

Solving a set C of constraints of the form $\tau_1 = \tau_2$ either *fails* or produces a *unifier*, *i.e.*, a type substitution σ which, when applied to all equations in C makes, each of the equations in C manifestly true: for all $(\tau_1 = \tau_2) \in C$, $\sigma(\tau_1)$ is syntactically equal to $\sigma(\tau_2)$.

The procedure for calculating the unifier (in fact: the *most general* unifier—the one that makes the least commitments) is called *unification* and goes back to Robinson (1971). The following version is from Pierce's textbook:

$$\begin{array}{rl} \mathit{unify}(C) = & \text{if } C = \emptyset \text{ then } [\,] \\ & \text{else let } \{\tau_1 = \tau_2\} \uplus C' = C \text{ in} \\ & \quad \text{if } \tau_1 = \tau_2 \\ & \quad\quad \text{then } \mathit{unify}(C') \\ & \quad \text{else if } \tau_1 = \alpha \wedge \alpha \notin \mathrm{FTV}(\tau_2) \\ & \quad\quad \text{then } \mathit{unify}(\{\tau_2/\alpha\}C') \circ [\alpha \mapsto \tau_2] \\ & \quad \text{else if } \tau - 2 = \alpha \wedge \alpha \notin \mathrm{FTV}(\tau_1) \\ & \quad\quad \text{then } \mathit{unify}(\{\tau_1/\alpha\}C') \circ [\alpha \mapsto \tau_1] \\ & \quad \text{else if } \tau_1 = \tau_{11} \to \tau_{12} \wedge \tau_2 = \tau_{21} \to \tau_{22} \\ & \quad\quad \text{then } \mathit{unify}(C' \cup \{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\}) \\ & \quad \text{else fail} \end{array}$$

7.3 ML-style "let"-polymorphism

Now consider adding the familiar **let**-construct to our language:

$$e \quad ::= \quad \ldots \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

We would like to be able to type-check the following program:

$$\textbf{let}\ f = \lambda x.x\ \textbf{in if}\ f\ \textbf{true then}\ 1 + f(2)\ \textbf{else}\ 0$$

For this, the two uses of f within the body of the **let**-form have to be assigned different types. The perhaps easiest way of doing this is to use the fact that evaluation of any **let** $x = e_1$ **in** e_2 substitutes (the result of) e_1 into e_2. Thus, the trick is to do the same within the typing rule. Using the declarative version of the rules (without type variables), we get:

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma \vdash \{e_1/x\}e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

The heart of this rule is its second premise; the first premise is added just to guarantee that e_1 has at least *some* type, even if x does not occur in e_2.

One problem with this rule is that it is unsound if we use it in a setting where e_1 can have side effects. Recall that substituting e_1 into e_2 is not what the true underlying dynamic semantics does (assuming we are in a CBV setting)! An effect in e_1 could be duplicated by the substitution, and this duplication might be safe while the original non-duplicated code is not. As a result, the above typing rule might be applicable in a situation where the dynamic semantics would get stuck. This situation is best illustrated by the following fragment of ML code:

```
let val r = ref (fn x => x)
in  r := (fn x => x+1);
    if !r true then "this" else "crashes"
end
```

While substituting `ref (fn x => x)` for the occurrences of `r` is indeed safe and will type-check (even though it might not be considered a very useful program), the original ML program is not sound because it would invoke the successor function on type `int` with a boolean argument.

One solution (and perhaps the simplest one) to this problem is to make sure that e_1 is substituted into e_2 only when it is a syntactic value. We can express this so-called *value restriction* using two separate typing rules:

$$\frac{\Gamma \vdash_{\mathrm{v}} v_1 : \tau' \qquad \Gamma \vdash \{v_1/x\}e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2}$$

7.4 Inferring polymorphic types

Another problem with the above rule for typing **let** is its potential inefficiency if used in a type-checking algorithm. After all, substituting e_1 into e_2 can duplicate e_1, and if this happens repeatedly, the size of the code that the type checker has to deal with explodes.

The idea for addressing this efficiency problem is to rely on the principal type of e_1: If the various copies of e_1 that are substituted into e_2 get different types, then all these types are still instances of e_1's principal type. Thus, it is more efficient to first calculate the principal type $\hat{\tau}_1$ of e_1, record a binding for x to $\hat{\tau}_1$ in the typing environment, and then use that to type-check e_2. At use-sites of x in e_2 the type checker will encounter $\hat{\tau}_1$ and can now "simply" instantiate it as needed.

Working this idea into a set of rules that amount to an algorithm starts with the constraint-based semantics that we have seen before. However, unlike before, we now use *unify* to solve portions of the constraints locally, as part of the rule for **let**. Unfortunately, the details get rather messy at this point since constraint-solving is now interleaved with the process of finding the derivation of the constraint-typing relation. For a complete account, see the literature (*e.g.*, starting with Pierce's textbook).

7.5 A declarative type system for let-polymorphism

A somewhat simpler way of specifying **let**-polymorphism without relying on substitution within the typing rules is to again use a declarative, non-deterministic set of typing rules. The idea is to "guess" at the time when e_1 is about to be type-checked what the free variables in the principal type of e_1 will be. These variables are added to a kinding environment Δ which is carried through the typing derivation just like it had been done in the type system for System F. Formally, Δ restricts the set of type variables that can be used when "guessing" other types at any given point.

Another change is that the typing environment Γ now has to bind variables to those principal types. Since we have to distinguish between type variables that are truly free and those that are actually introduced and bound "higher up" for a **let** that is in the surrounding context, we explicitly quantify over the free variables using the familiar $\forall$ binder. This gives rise to *polymorphic* types schemata σ:

types:	τ	$::=$	$\alpha \mid$	type variables
			$\tau \to \tau \mid$	function types
			b	some base type(s)
schemata:	σ	$::=$	$\tau \mid$	trivial schema
			$\forall\alpha.\ \sigma$	polymorphic type
values:	v	$::=$	$x \mid$	variables
			$\lambda x.e \mid$	abstractions
			c_τ	constants (of type τ)
expressions:	e	$::=$	$v \mid$	values
			$e\ e \mid$	applications
			$\mathbf{let}\ x = e\ \mathbf{in}\ e$	let-binding
kinding environment:	Δ	$\subseteq$	TyVar	sets of type variables
typing environments:	Γ	$\in$	$\mathrm{Var} \mapsto^{\mathrm{fin}} \sigma$	finite mappings from variables to types

The type system consists of several sets of rules.

Well-formedness of types: $\boxed{\Delta \vdash \tau\ \mathbf{ok}}$ These rules work basically the same way as they did in case of System F. However, since the only place where $\forall$-binders occur is Γ, the rules never have to "guess" a type that is polymorphic. In this sense polymorphic types are not considered well-formed.

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathbf{ok}}\ \textsc{tyvar} \qquad \frac{}{\Delta \vdash b\ \mathbf{ok}}\ \textsc{basety} \qquad \frac{\Delta \vdash \tau_1\ \mathbf{ok} \qquad \Delta \vdash \tau_2\ \mathbf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \mathbf{ok}}\ \textsc{funty}$$

Typing of variables: $\boxed{\Delta; \Gamma \vdash_{\mathrm{x}} x : \sigma}$ The idea here is that the type of a variable starts out as a polymorphic type schema σ, namely the one recorded in Γ. This polymorphic type schema then gets "boiled down" to an ordinary monomorphic type by successive instantiation of the quantified type variables:

$$\frac{\Gamma(x) = \sigma}{\Delta; \Gamma \vdash_{\mathrm{x}} x : \sigma}\ \textsc{lookup-p} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{x}} x : \forall\alpha.\ \sigma \qquad \Delta \vdash \tau\ \mathbf{ok}}{\Delta; \Gamma \vdash_{\mathrm{x}} x : \{\tau/\alpha\}\sigma}\ \textsc{inst-p}$$

Typing of values: $\boxed{\Delta; \Gamma \vdash_{\mathrm{v}} v : \tau}$ The most important new part here is the typing of variables: the rule relies on the variable typing judgment from above, but at the same time insists that the type of the variable be an ordinary type (τ) and not some truly polymorphic type schema ($\forall\alpha.\ \sigma$). The second change concerns the choice of τ_1 in the rule for λ-terms: whatever is chosen must be well-formed in the current kinding context:

$$\frac{\Delta; \Gamma \vdash_{\mathrm{x}} x : \tau}{\Delta; \Gamma \vdash_{\mathrm{v}} x : \tau}\ \textsc{var-p} \qquad \frac{\Delta \vdash \tau_1\ \mathbf{ok} \qquad \Delta; \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Delta; \Gamma \vdash_{\mathrm{v}} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{lam-p}$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{v}} c_\tau : \tau}\ \textsc{con-p}$$

Typing of expressions: $\boxed{\Delta;\Gamma \vdash e : \tau}$ The rules for values and applications are standard. Our focus is on the two rules that concern **let**-expressions—one for situations where the value restriction is satisfied and polymorphic generalization can be applied (LETV-P) and one for other **let**-forms where the type of e_1 cannot be made polymorphic (LET-P):

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} v : \tau}{\Delta;\Gamma \vdash v : \tau}\ \textsc{val-p} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{app-p}$$

$$\frac{\begin{array}{c}\alpha_1,\ldots,\alpha_k \notin \Delta \qquad \Delta \cup \{\alpha_1,\ldots,\alpha_k\};\Gamma \vdash_{\mathrm{v}} v_1 : \tau_1 \\ \sigma_1 = \forall\alpha_1.\ \ldots \forall\alpha_k.\ \tau_1 \qquad \Delta;\Gamma[x \mapsto \sigma_1] \vdash e_2 : \tau_2\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = v_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsc{letv-p}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \qquad \Delta;\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsc{let-p}$$

7.6 Elaboration into System F

The declarative typing rules given in the previous section more than just superficially resemble those for System F. Of course, in the case of System F we have a deterministic type system because the underlying language is *explicit* in where polymorphism is introduced (namely at type abstractions $\Lambda\alpha.\ e$) and where it is eliminated (namely at type applications $e[\tau]$). On the other hand, in the *implicitly* typed language with **let**-polymorphism, the typing derivation determines where polymorphism is to be introduced and where it is to be eliminated.

However, the beauty of the correspondence is that we can extend the rules for type inference and turn them into rules for *elaboration*. Elaboration *translates* implicitly typed expressions of our "source" language into explicitly typed System F terms. Since System F already comes with a dynamic semantics, this approach can be used to give a dynamic semantics for the source language *by way of elaboration*.[11]

An elaboration judgment has the form $\Delta;\Gamma \vdash e : \tau \rightsquigarrow \hat{e}$. The part up to $\rightsquigarrow$ is identical to the original typing judgment. The term $\hat{e}$ is the translation term. Its syntax is governed by the rules of the target language, *i.e.*, System F. (There is an analogous translation judgment for values and also one for variables.)

We start by looking at the translation of variables. Since the typing of variables involves polymorphic instantiation, the translation must make these instantiations explicit by producing the corresponding type applications:

[11] It should be noted that it can be quite non-trivial to prove that the elaboration semantics is equivalent to the original semantics. Indeed, since the typing rules—and, thus, the elaboration rules—are non-deterministic in nature, there can be more than one valid elaboration term. To show that any two elaboration terms of the same source term are (observationally) equivalent to each other is the problem of *coherence*.

$$\frac{\Gamma(x) = \sigma}{\Delta;\Gamma \vdash_{\mathrm{x}} x : \sigma \rightsquigarrow x}\ \textsc{lookup-e} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{x}} x : \forall\alpha.\ \sigma \rightsquigarrow \hat{e} \qquad \Delta \vdash \tau\ \mathbf{ok}}{\Delta;\Gamma \vdash_{\mathrm{x}} x : \{\tau/\alpha\}\sigma \rightsquigarrow \hat{e}[\tau]}\ \textsc{inst-e}$$

Next, we consider the translation of values, which is straightforward:

$$\frac{\Delta;\Gamma \vdash_{\mathrm{x}} x : \tau \rightsquigarrow \hat{e}}{\Delta;\Gamma \vdash_{\mathrm{v}} x : \tau \rightsquigarrow \hat{e}}\ \textsc{var-e} \qquad \frac{\Delta \vdash \tau_1\ \mathbf{ok} \qquad \Delta;\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \rightsquigarrow \hat{e}}{\Delta;\Gamma \vdash_{\mathrm{v}} \lambda x.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \tau_1.\ \hat{e}}\ \textsc{lam-e}$$

$$\frac{}{\Delta;\Gamma \vdash_{\mathrm{v}} c_\tau : \tau \rightsquigarrow c_\tau}\ \textsc{con-e}$$

The translation of expressions is also straightforward—of course with the exception of **let**. If a **let**-bound variable is assigned a polymorphic type, the System F translation must introduce a corresponding type abstraction. Notice that the **let**-form within the translation is merely syntactic sugar for an application where the operator is a λ-term:

$$\frac{\Delta;\Gamma \vdash_{\mathrm{v}} v : \tau \rightsquigarrow \hat{e}}{\Delta;\Gamma \vdash v : \tau \rightsquigarrow \hat{e}}\ \textsc{val-e} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau \rightsquigarrow \hat{e}_1 \qquad \Delta;\Gamma \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau \rightsquigarrow \hat{e}_1\ \hat{e}_2}\ \textsc{app-p}$$

$$\frac{\begin{array}{c}\alpha_1,\ldots,\alpha_k \notin \Delta \qquad \Delta \cup \{\alpha_1,\ldots,\alpha_k\};\Gamma \vdash_{\mathrm{v}} v_1 : \tau_1 \rightsquigarrow \hat{e}_1 \\ \sigma_1 = \forall\alpha_1.\ \ldots\forall\alpha_k.\ \tau_1 \qquad \Delta;\Gamma[x \mapsto \sigma_1] \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = v_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x = \Lambda\alpha_1.\ldots\Lambda\alpha_k.\ \hat{e}_1\ \mathbf{in}\ \hat{e}_2}\ \textsc{letv-p}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \rightsquigarrow \hat{e}_1 \qquad \Delta;\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2 \rightsquigarrow \hat{e}_2}{\Delta;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x = \hat{e}_1\ \mathbf{in}\ \hat{e}_2}\ \textsc{let-p}$$

Lemma 7.1
The translation is type-correct in the sense that if $\Delta;\Gamma \vdash e : \tau$*, then there exists some* $\hat{e}$ *such that* $\Delta;\Gamma \vdash e : \tau \rightsquigarrow \hat{e}$*. Moreover, in this case we also have* $\Delta;\Gamma \vdash \hat{e} : \tau$ *(using the typing rules for System F).*

The proof for this lemma proceeds by straightforward induction on the derivation of $\Delta;\Gamma \vdash e : \tau$. A corollary of this result is the following: If we take the dynamic semantics of the source language as being defined via elaboration, then the source language is safe—simply because well-typed source terms translate to well-typed System F terms, and System F itself is already known to be safe.

Notice that the above lemma together with the associated safety result would be true *even if we did not insist on the value restriction* in the rules for **let**! How can this be? Did we not say that the value restriction is needed for soundness? The explanation for this mystery is that inference of a polymorphic type goes hand-in-hand with the introduction of corresponding Λ-binders in the translation. Thus, any computation expressed by e_1 in $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ gets *suspended*,

and every use of x within e_2 re-plays this computation. Thus, the target term behaves as if the **let**-expression had been translated by actually substituting e_1 for x in e_2.

Problems with soundness as shown in the example above stem from the fact that the typing rules behave *as if* e_1 gets suspended until needed by e_2 (at which time it gets re-evaluated every time e_2 touches x), while the dynamic semantics that is actually used by an implementation does not behave that way. Now, as we have seen, our elaboration-based semantics *does* behave that way, so the typing rule without the value restriction is sound here. However, such behavior is not the intended and expected one: we *do* want e_1 fully evaluated once and for all before e_2 starts executing, and the introduction of polymorphism should not change that. Thus, we need a way of preventing polymorphic generalization when e_1 is not pure. The value restriction does the job.

7.7 Imperative type inference

An elegant method for implementing type inference is to solve constraints immediately as they arise. This, combined with the use of the *mutable store* of an imperative programming language leads to a very compact version of the algorithm. The idea here is to implement type variables as *references* to type *options*, where `ref NONE` stands for an unconstrained type variable and `ref (SOME` t`)` stands for a type variable that is constrained to be equal to type t.

Concretely, let us use the following ML datatype to represent types:

```
datatype typ =
      INTt
    | BOOLt
    | FUNt of typ * typ
    | VARt of tyvar

withtype tyvar = typ option ref
```

The following unification procedure `unify` either enforces that two types be equal or otherwise fails:

```
fun unify (INTt, INTt) = ()
  | unify (BOOLt, BOOLt) = ()
  | unify (FUNt (t1, t2), FUNt (t1', t2')) =
      (unify (t1, t1'); unify (t2, t2'))
  | unify (VARt (ref (SOME t1)), t2) = unify (t1, t2)
  | unify (t1, VARt (ref (SOME t2))) = unify (t1, t2)
  | unify (t1 as VARt (r1 as ref NONE),
           t2 as VARt (r2 as ref NONE)) =
      if r1 = r2 then () else (r1 := SOME t2)
  | unify (t1 as VARt (r1 as ref NONE), t2) =
      if occurs (r1, t2) then raise Fail "circular type"
      else r1 := SOME t2
```

```
  | unify (t1, t2 as VARt (r2 as ref NONE)) =
      if occurs (r2, t1) then raise Fail "circular type"
      else r2 := SOME t1
  | unify _ = raise Fail "type mismatch"
```

Notice the use of the helper function `occurs` in this code. This function implements the so-called *occur check*, which makes sure that we do not create cyclic structures which would correspond to "infinite" types. Generating cyclic type structures does not correspond to anything in the type system that we have seen so far, but even more seriously, it would also lead to potential non-termination in the type checker itself. For example, if t equals `ref(SOME` t`)`, then `unify(`t`,`t`)` would loop forever.

The occur check itself is implemented as follows:

```
fun occurs (r, INTt) = false
  | occurs (r, BOOLt) = false
  | occurs (r, FUNt (t1, t2)) = occurs (r, t1) orelse occurs (r, t2)
  | occurs (r, VARt r') =
      r = r' orelse
      (case !r' of
          NONE => false
        | SOME t => occurs (r, t))
```

With this preparation in place, the type inference algorithm itself follows almost directly the pattern given by the *declarative* type system. The idea is to have functions `etyp` and `vtyp` which "implement" the judgments $\Gamma \vdash e : \tau$ and $\Gamma \vdash_{\mathrm{v}} v : \tau$, respectively, by taking an argument `G` (for Γ), an argument `e` or `v` (for e or v), and an argument `t` (for τ). Instead of calculating the type of an expression or a value, the functions enforce that `t` be the type of `e` or `v` under the typing environment `G`. Of course, `t` can be a fresh type variable (*i.e.*, `VARt (ref NONE)`), in which case the above enforcement will effectively calculate the type in question and store it into the `ref`-cell in `t`.

```
fun newtyp () = VARt (ref NONE)
```

To illustrate the idea, here are the cases for λ-terms and application:

```
fun vtyp ... = ...  (* other cases *)
  | vtyp (G, FUN { f, x, body }, t) =
      let val t1 = newtyp ()          (* argument type *)
          val t2 = newtyp ()          (* result type *)
      in etyp (bind (t, f, bind (t1, x, G)), body, t2);
                                      (* G,f:t,x:t1 |- body : t2 *)
         unify (t, FUNt (t1, t2))     (* t = t1 -> t2 *)
      end
and etyp ... = ...  (* other cases *)
  | etyp (G, APPLY (e1, e2), t) =
```

```
let val t2 = newtyp ()
in etyp (G, e1, FUNt (t2, t));  (* G |- e1 : t2 -> t *)
   etyp (G, e2, t2)             (* G |- e2 : t2 *)
end
```

Imperative type inference for ML

The approach shown above makes it easy to implement type inference for the Simply Typed λ-calculus. It can, however, also be strengthened to deal with ML-style polymorphism, which in the case of expressions of the form **let** $x = v_1$ **in** e_2 infers universally quantified types for v_1 to be used within e_2.

For this, ML-style polymorphic type inference must partially solve the constraints that involve v_1 before proceeding to type-check e_2. The good news is that the imperative algorithm already solves all constraints eagerly, so no additional solving needs to be performed by the **let**-rule. The idea is to simply pick out the unconstrained type variables in the type of v_1 and put a $\forall$ quantifier on them. Some care is necessary, though: not every type of the form `VARt(ref NONE)` is conceptually unconstrained. If such a variable occurs in the typing environment `G` (*a.k.a.*, Γ), then it must be considered constrained after all.

Since checking for the presence of a type variable in `G` can be expensive, the trick is to keep a numeric level with each not-yet-instantiated type variable. The level indicates the earliest position within `G` where this variable occurs. When new type variables are generated, the level is initialized to the *current level* (which can be though of as the size of the domain of `G` but which is usually carried explicitly through the type checker as an additional argument). But when a type variable of level l is instantiated to type t, the levels of variables within t must be adjusted to be no later than l. The traversal of t on behalf of the occur check can do double-duty to implement this adjustment. Any type variable in the type of v_1 that has not been instantiated and whose level is greater or equal that the current level can be universally quantified.

8 Recursive types

With some algorithmic changes to prevent `unify` from chasing down infinite (type-)loops, it can make sense to leave out the occur check from the type inference algorithm. The result, however, is the possibility of cyclic data structures that supposedly represent certain types. If we allow that, we have to formally account for these types.

A cyclic type structure can be thought of as a compact representation of an *infinite* type. If we have a type t that is equal to `FUNt (VARt (ref t), INTt)`, then t represents the type

$$((((\ldots \to \mathbf{int}) \to \mathbf{int}) \to \mathbf{int}) \to \mathbf{int}$$

which is a solution to the type equation

$$\alpha = \alpha \to \mathbf{int}.$$

However, to handle such types as part of our type system, we need a way of representing them as *finite* terms! Thus, we add a new construct, called a *recursive type*. Recursive types are written $\mu\alpha.\tau$ where μ is a binder (much like $\forall$, $\exists$, or Λ), where α is a type variable, and where τ is a type that may or may not mention α. If the type language did not already have type variables, then we would have to add those as well. (If we start with System F, we can simply use the type variables that are already present in the language.) Of course, we have to account for μ by adding a new kinding rule:

$$\frac{\Delta, \alpha \vdash \tau \ \mathbf{ok} \qquad \alpha \notin \Delta}{\Delta \vdash \mu\alpha.\tau \ \mathbf{ok}}$$

The idea is that the binder for α "marks" the current type, and any use of α within τ serves as a back-reference to the marked type.

8.1 Equi-recursive types

There are (at least) two ways of thinking about recursive types. The first is to postulate that $\mu\alpha.\tau$ is the solution to the type equation $\alpha = \tau$. That is, within its scope, α can be replaced with $\mu\alpha.\tau$ without changing the meaning of the type. And, since α is not longer free in $\{(\mu\alpha.\tau)/\alpha\}\tau$, the outer μ-binder can be removed, giving rise to the equation:

$$\mu\alpha.\tau = \{(\mu\alpha.\tau)/\alpha\}\tau$$

Having such an equation means that types can be considered equivalent even if they are not syntactically identical. We can account for such additional equivalences by either having an explicit equational theory for types (*i.e.*, rules for saying when types are equivalent), or we can add typing rules that let us switch between the left- and right-hand sides of the above equation:

$$\frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau}{\Delta; \Gamma \vdash e : \{(\mu\alpha.\tau)/\alpha\}\tau}\ \textsc{implicit-unroll} \qquad \frac{\Delta; \Gamma \vdash e : \{(\mu\alpha.\tau)/\alpha\}\tau}{\Delta; \Gamma \vdash e : \mu\alpha.\tau}\ \textsc{implicit-roll}$$

Since this approach treats $\mu\alpha.\tau$ and $\{(\mu\alpha.\tau)/\alpha\}\tau$ equivalently, it is referred to as *equi-recursive types*. Since in either rule, Δ, Γ, and e are the same in both the premise and the conclusion, their addition means that the typing rules are no longer *structural*, *i.e.*, they no longer strictly follow the syntactic composition of the underlying expression.

8.2 Iso-recursive types

The other approach, called *iso-recursive types*, treats $\mu\alpha.\tau$ as an "abstract" type with explicit operations **roll** and **unroll**[12] for converting from and to

[12] Some authors call these operations **fold** and **unfold**.

$\{(\mu\alpha.\tau)/\alpha\}\tau$. The name comes from the fact that these two operations establish an isomorphism between the two types.

$$e \quad ::= \quad \ldots \mid \mathbf{roll}[\mu\alpha.\tau]\ e \mid \mathbf{unroll}\ e$$

The typing rules mirror the two rules from the equi-recursive account above, the difference being the fact that they are still structural rules:

$$\frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau}{\Delta; \Gamma \vdash \mathbf{unroll}\ e : \{(\mu\alpha.\tau)/\alpha\}\tau}\ \text{EXPLICIT-UNROLL}$$

$$\frac{\Delta; \Gamma \vdash e : \{(\mu\alpha.\tau)/\alpha\}\tau}{\Delta; \Gamma \vdash \mathbf{roll}[\mu\alpha.\tau]e : \mu\alpha.\tau}\ \text{EXPLICIT-ROLL}$$

Notice that **roll** is annotated with the recursive type to make the typing rule deterministic as otherwise there can be more than one way of picking a τ and an α in such a way that the type of e is $\{(\mu\alpha.\tau)/\alpha\}\tau$.

The operational semantics of **roll** and **unroll** is, simply speaking, that **unroll** cancels **roll** in the following sense: first we add $\mathbf{fold}[\mu\alpha.\tau]v$ to the language of values. Then we add these small-step rules:

$$\frac{e \mapsto^1 e'}{\mathbf{roll}[\mu\alpha.\tau]e \mapsto^1 \mathbf{roll}[\mu\alpha.\tau]e'} \qquad \frac{e \mapsto^1 e'}{\mathbf{unroll}\ e \mapsto^1 \mathbf{unroll}\ e'}$$

$$\frac{}{\mathbf{unroll}\ (\mathbf{roll}[\mu\alpha.\tau]v) \mapsto^1 v}$$

An alternative (and ultimately equally expressive) design is to make **roll** a "suspending" construct by letting $\mathbf{roll}[\mu\alpha.\tau]e$ be a value. Then the first structural rule of the small-step semantics would be eliminated, and **unroll** would have the effect of un-suspending a computation previously suspended via **roll**.

8.3 General recursion from recursive types

As we have seen, pure System F is a fairly powerful language. However, it is not Turing-complete, as all its programs strictly terminate. The addition of recursive types pushes the language over the top: it adds the ability to perform arbitrary recursion and makes the language Turing-complete. This also means that it is no longer true that all programs terminate.

In the untyped CBV λ-calculus, we can use the Y combinator to define arbitrary recursive functions:

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

In System F there was no way of assigning a type to this term, because of the self-application of x. With recursive types, however, it is now possible to assign a type:

$$Y = \Lambda\alpha\Lambda\beta.\ \lambda f : (\alpha \to \beta) \to (\alpha \to \beta).\ (B\ (\mathbf{roll}[\mu\gamma.\ (\gamma \to \alpha \to \beta)]B))$$

where B is the following term:

$$B = \lambda x : \mu\gamma.(\gamma \to \alpha \to \beta).\ f\ (\lambda y : \alpha.\ (\mathbf{unroll}\ x)\ x\ y).$$

It is easy to check that this term is well-typed under the iso-recursive account. Its type is

$$Y : \forall\alpha\forall\beta.\ ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

To get the equi-recursive version, it suffices to delete the $\mathbf{roll}[\cdots]$ and **unroll** parts from the expression.

9 Existential types

Augment System F as follows:

$$\begin{array}{lcl} \tau & ::= & \ldots \mid \exists\alpha.\tau \\ v & ::= & \ldots \mid \mathbf{pack}\ [\tau, v]\ \mathbf{as}\ \exists\alpha.\tau \\ e & ::= & \ldots \mid \mathbf{pack}\ [\tau, e]\ \mathbf{as}\ \exists\alpha.\tau \mid \mathbf{unpack}\ [\alpha, x] = e\ \mathbf{in}\ e \end{array}$$

9.1 Well-formedness of types

$$\frac{\alpha \notin \Delta \qquad \Delta, \alpha \vdash \tau\ \mathbf{ok}}{\Delta \vdash \exists\alpha.\tau\ \mathbf{ok}}$$

9.2 Typing rules

$$\frac{\Delta \vdash \exists\alpha.\tau\ \mathbf{ok} \qquad \Delta; \Gamma \vdash e : \{\sigma/\alpha\}\tau}{\Delta; \Gamma \vdash \mathbf{pack}\ [\sigma, e]\ \mathbf{as}\ \exists\alpha.\tau : \exists\alpha.\tau}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \exists\beta.\tau \qquad \Delta \vdash \tau_2\ \mathbf{ok} \qquad \alpha \notin \Delta \qquad \Delta, \alpha; \Gamma[x \mapsto \{\alpha/\beta\}\tau] \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \mathbf{unpack}\ [\alpha, x] = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

9.3 Structural operational semantics (small-step)

$$\frac{e \mapsto^1 e'}{\textbf{pack}\ [\sigma, e]\ \textbf{as}\ \exists\alpha.\tau \mapsto^1 \textbf{pack}\ [\sigma, e']\ \textbf{as}\ \exists\alpha.\tau}$$

$$\frac{e_1 \mapsto^1 e_1'}{\textbf{unpack}\ [\alpha, x] = e_1\ \textbf{in}\ e_2 \mapsto^1 \textbf{unpack}\ [\alpha, x] = e_1'\ \textbf{in}\ e_2 \mapsto^1}$$

$$\frac{}{\textbf{unpack}\ [\alpha, x] = (\textbf{pack}\ [\sigma, v]\ \textbf{as}\ \exists\beta.\tau)\ \textbf{in}\ e_2 \mapsto^1 \{\sigma/\alpha\}(\{v/x\}e_2)}$$

9.4 Church-encoding existential types

$$\begin{array}{rcl} \exists\beta.\tau & \equiv & \forall\alpha.\ (\forall\beta.\ \tau \to \alpha) \to \alpha \\ \textbf{pack}\ [\sigma, e]\ \textbf{as}\ \exists\beta.\tau & \equiv & \Lambda\alpha.\ \lambda k : \forall\beta.\ \tau \to \alpha.\ k[\sigma]\ e \\ \textbf{unpack}\ [\alpha, x] = e_1\ \textbf{in}\ e_2 & \equiv & e_1[\tau_2]\ (\Lambda\alpha.\ \lambda x : \tau.\ e_2) \end{array}$$