Lecture Notes on Inductive Definitions

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 2
September 2, 2004

These supplementary notes review the notion of an inductive definition and give some examples of rule induction. References to Robert Harper's draft book on *Programming Languages: Theory and Practice* are given in square brackets, by chapter or section.

Given our general goal to define and reason about programming languages, we will have to deal with a variety of description tasks. The first is to describe the grammar of a language. The second is to describe its static semantics, usually via some typing rules. The third is to describe its dynamic semantics, often via transitions of an abstract machine. On the surface, these appear like very different formalisms (grammars, typing rules, abstract machines) but it turns out that they can all be viewed as special cases of *inductive definitions* [Ch. 1]. Following standard practice, inductive definitions will be presented via judgments and inference rules providing evidence for judgments.

The first observation is that context-free grammars can be rewritten in the form of inference rules [Ch. 3.2]. The basic judgment has the form

$$s\ A$$

where s is a string and A is a non-terminal. This should be read as the judgment that *s is a string of syntactic category A*.

As a simple example we consider the language of properly matched parentheses over the alphabet $\Sigma = \{\mathsf{(}, \mathsf{)}\}$. This language can be defined by the grammar

$$M ::= \varepsilon \mid \mathsf{(}\, M \,\mathsf{)} \mid M\, M$$

with the only non-terminal M. Recall that ε stands for the empty string.

Rewritten as inference rules we have:

$$\frac{}{\varepsilon\ M}\ m_1 \qquad \frac{s\ M}{(\,s\,)\ M}\ m_2 \qquad \frac{s_1\ M \qquad s_2\ M}{s_1\ s_2\ M}\ m_3$$

As an example, consider a deduction of $(\,)\ (\,)\ M$.

$$\dfrac{\dfrac{\dfrac{}{\varepsilon\ M}\ m_1}{(\,)\ M}\ m_2 \qquad \dfrac{\dfrac{}{\varepsilon\ M}\ m_1}{(\,)\ M}\ m_2}{(\,)\ (\,)\ M}\ m_3$$

Our interpretation of these inference rules as an inductive definition of the judgment $s\ M$ for a string s means:

> *$s\ M$ holds if and only if there is a deduction of $s\ M$ using rules (m_1), (m_2), and (m_3).*

Based on this interpretation we can prove properties of strings in the syntactic category M by rule induction. To apply rule induction we have to show that the property in question is preserved by every inference rule of the judgment $s\ M$. That is, we have to show that for each rule, if all premises satisfy the property then the conclusion also satisfies the property. Here is a very simple example.

Theorem 1 (Counting Parentheses)
If $s\ M$ then s has the same number of left and right parentheses.

Proof: By rule induction. We consider each case in turn.

(Rule m_1) Then $s = \varepsilon$.

s has 0 left and 0 right parens	Since $s = \varepsilon$

(Rule m_2) Then $s = (\,s'\,)$.

$s'\ M$	Subderivation
s' has n' left and right parens for some n'	By i.h.
s has $n' + 1$ left and right parens	Since $s = (\,s'\,)$

(Rule m_3) Then $s = s_1\, s_2$.

$s_1\ M$	Subderivation
$s_2\ M$	Subderivation
s_1 has n_1 left and right parens for some n_1	By i.h.
s_2 has n_2 left and right parens for some n_2	By i.h.
s has $n_1 + n_2$ left and right parens	Since $s = s_1\, s_2$

■

The grammar we gave, unfortunately, is ambiguous [Ch. 3.3]. For example, there are infinitely many derivations that $\varepsilon\ M$, because

$$\varepsilon = \varepsilon\,\varepsilon = \varepsilon\,\varepsilon\,\varepsilon = \cdots$$

In the particular example of this grammar we would be able to avoid rewriting it if we can show that the abstract syntax tree [Ch. 4] we construct will be the same, independently of the derivation of a particular judgment.

An alternative is to rewrite the grammar so that it defines the same language of strings, but the derivation of any particular string is uniquely determined. The following grammar accomplishes this: [1]

$$L \quad ::= \quad \varepsilon \mid (\,L\,)\,L$$

One can think of L as a (possibly empty) list of parenthesized expressions, terminated by the empty string. This readily translates into an inductive definition via inference rules.

$$\frac{}{\varepsilon\ L}\ l_1 \qquad \frac{s_1\ L \qquad s_2\ L}{(\,s_1\,)\ s_2\ L}\ l_2$$

Now there are two important questions to ask: (1) is the new grammar really equivalent to the old one in the sense that it generates the same set of

[1] An alternative solution, suggested in lecture in 2003, exemplifies the idea of a simultaneous inductive definition. It uses two non-terminals L and N, where the category L corresponds to M, while N is an auxiliary non-terminal.

$$\begin{array}{lcl} L & ::= & \varepsilon \mid N\,L \\ N & ::= & (\,L\,) \end{array}$$

Note that the new definition arises from substituting out the definition of N in the alternation for L.

strings, and (2) is the new grammar really unambiguous. The latter is left as a (non-trivial!) exercise; the first one we discuss here.

At a high level we want to show that for any string s, $s\ M$ iff $s\ L$. We break this down into two lemmas. This is because "if-and-only-if" statements can rarely be proven by a single induction, but require different considerations for the two directions.

We first consider the direction where we assume $s\ M$ and try to show $s\ L$. When writing out the cases we notice we need an additional lemma. As is often the case, the presentation of the proof is therefore different from its order of discovery. To read this proof in a more natural order, skip ahead to Lemma 3 and pay particular attention to the last step in the case of rule (m_3). That step motivates the following lemma.

Lemma 2 (Concatenation)
If $s_1\ L$ and $s_2\ L$ then $s_1\ s_2\ L$.

Proof: By induction on the derivation of $s_1\ L$. Note that induction on the derivation on $s_2\ L$ will not work in this case!

(Rule l_1) Then $s_1 = \varepsilon$.

$s_2\ L$	Assumption
$s_1\ s_2\ L$	Since $s_1\ s_2 = \varepsilon\ s_2 = s_2$

(Rule l_2) Then $s_1 = (\ s_{11}\ )\ s_{12}$.

$s_{11}\ L$	Subderivation
$s_{12}\ L$	Subderivation
$s_2\ L$	Assumption
$s_{12}\ s_2\ L$	By i.h.
$(\ s_{11}\ )\ s_{12}\ s_2\ L$	By rule (l_2)

■

Now we are ready to prove the left-to-right implication.

Lemma 3
If $s\ M$ then $s\ L$.

Proof: By induction on the derivation of $s\ M$.

(Rule m_1) Then $s = \varepsilon$.

$s\ L$	By rule (l_1) since $s = \varepsilon$

(Rule m_2) Then $s = (\ s'\ )$.

$s'\ M$	Subderivation
$s'\ L$	By i.h.
$\varepsilon\ L$	By rule (l_1)
$(\ s'\ )\ L$	By rule (l_2) and $(\ s'\ )\ \varepsilon = (\ s'\ )$

(Rule m_3) Then $s = s_1\ s_2$.

$s_1\ M$	Subderivation
$s_2\ M$	Subderivation
$s_1\ L$	By i.h.
$s_2\ L$	By i.h.
$s_1\ s_2\ L$	By concatenation (Lemma 2)

■

The right-to-left direction presents fewer problems.

Lemma 4
If $s\ L$ then $s\ M$.

Proof: By rule induction on the derivation of $s\ L$. There are two cases to consider.

(Rule l_1) Then $s = \varepsilon$.

$s\ M$	By rule (m_1), since $s = \varepsilon$

(Rule l_2) Then $s = (\ s_1\ )\ s_2$.

$s_1\ L$	Subderivation
$s_2\ L$	Subderivation
$s_1\ M$	By i.h.
$(\ s_1\ )\ M$	By rule (m_2)
$s_2\ M$	By i.h.
$(\ s_1\ )\ s_2\ M$	By rule (m_3)

■

Now we can combine the preceding lemmas into the theorem we were aiming for.

Theorem 5
$s\ M$ if and only if $s\ L$.

Proof: Immediate from Lemmas 3 and 4. ■

Some advice on inductive proofs. Most of the proofs that we will carry out in the class are by induction. This is simply due to the nature of the objects we study, which are generally defined inductively. Therefore, when presented with a conjecture that does not follow immediately from some lemmas, we first try to prove it by induction as given. This might involve a choice among several different given objects or derivations over which we may apply induction. If one of them works we are, of course, done. If not, we try to analyse the failure in order to decide if (a) we need to seperate out a *lemma* to be proven first, (b) we need to *generalize the induction hypothesis*, or (c) our conjecture might be false and we should look for a *counterexample*.

Finding a lemma is usually not too difficult, because it can be suggested by the gap in the proof attempt you find it impossible to fill. For example, in the proof of Lemma 3, case (Rule m_3), we obtain $s_1\ L$ and $s_2\ L$ by induction hypothesis and have to prove $s_1\ s_2\ L$. Since there are no inference rules that would allow such a step, but it seems true nonetheless, we prove it as Lemma 2.

Generalizing the induction hypothesis can be a very tricky balancing act. The problem is that in an inductive proof, the property we are trying to establish occurs twice: once as an inductive assumption and once as a conclusion we are trying to prove. If we strengthen the property, the induction hypothesis gives us more information, but conclusion becomes harder to prove. If we weaken the property, the induction hypothesis gives us less information, but the conclusion is easier to prove. Fortunately, there are easy cases in which the nature of the mutually recursive judgments suggested a generalization.

Finding a counterexample greatly varies in difficulty. Mostly, in this course, counterexamples only arise if there are glaring deficiencies in the inductive definitions, or rather obvious failure of properties such as type safety. In other cases it might require a very deep insight into the nature

of a particular inductive definition and cannot be gleaned directly from a failed proof attempt. An example of a difficult counterexample is given by the extra credit Question 2.2 in Assignment 1 of this course. The conjecture might be that every tautology is a theorem. However, there is very little in the statement of this theorem or in the definition of *tautology* and *theorem* which would suggest means to either prove or refute it.

Three pitfalls to avoid. The difficulty with inductive proofs is that one is often blinded by the fact that the proposed conjecture is true. Similarly, if set up correctly, it will be true that in each case the induction hypothesis does in fact imply the desired conclusion, but the induction hypothesis may not be strong enough to prove it. So you must avoid the temptation to declare something as "clearly true" and prove it instead.

The second kind of mistake in an inductive proof that one often encounters is a confusion about the direction of an inference rule. If you reason backwards from what you are trying to prove, you are thinking about the rules bottom up: "*If I only could prove J_1 then I could conclude J_2, because I have an inference rule with premise J_1 and conclusion J_2.*" Nonetheless, when you write down the proof in the end you must use the rule in the proper direction. If you reason forward from your assumptions using the inference rules top-down then no confusion can arise. The only exception is the proof principle of inversion, which you can *only* employ if (a) you have established that a derivation of a given judgment J exists, and (b) you consider all possible inference rules whose conclusion matches J. We will see examples of this form of reasoning later in the course. In no other case can you use an inference rule "backwards".

The third mistake to avoid is to apply the induction hypothesis to a derivation that is not a subderivation of the one you are given. Such reasoning is circular and unsound. You must always verify that when you claim something follows by induction hypothesis, it is in fact legal to apply it!

How much to write down. Finally, a word on the level of detail in the proofs we give and the proofs we expect you to provide in the homework assignments. The proofs in this handout are quite pedantic, but we ask you to be just as pedantic unless otherwise specified. In particular, you *must* show any lemmas you are using, and you *must* show the generalized induction hypothesis in an inductive proof (if you need a generalization). You also *must* consider all the cases and *justify each line* carefully. As we

gain a certain facility with such proofs, we may relax these requirements once we are certain you know how to fill in the steps that one might omit, for example, in a research paper.

Specifications vs. implementations. The grammar of our language of properly balanced parentheses (and also its formulation as an inductive definition) must be seen as a *specification*. That is, we define a language of strings (in the case of the grammar) or the judgment $s\ M$ (in the case of a judgment), but we do not immediately provide an *implementation*. In this case, such an implementation would be an algorithm for recognizing if a given string is a properly balanced string of parentheses. Ambiguity in the grammar, as noted in class, is one obstacle to deriving a parser from the specification of a grammar. In general, there are large classes of languages (including those specified by a context-free grammars) for which we can uniformly generate a parser from a grammar. Here, we will pursue a different direction, namely writing a parser for this specific language and proving that it is correct.

Interpreting inference rules as algorithms. To implement a parser, one would normally pick a programming language and simply write a program. However, then we would be faced with the problem of proving the correctness of that program, which depends on the details of the definition of the underlying implementation language.

Here we exploit instead that it is also possible to present some algorithms in the form of inference rules. Performing the algorithm corresponds to the search for a deduction of a judgment, as we will see shortly below. In programming language terminology this approach is called *logic programming*.

But first we have to decide on an algorithm for recognizing if a given string consists of properly matched parentheses. The informal idea of the parsing process for matching parentheses is quite straightforward: we keep an integer counter, initialized to zero, and increment it when we see an opening parenthesis and decrement it when we see a closing parenthesis. We need to check two conditions: (a) the counter never becomes negative (otherwise there would be too many closing parentheses) and (b) the counter is zero at the end (otherwise there would be unmatched open parentheses).

The process of parsing then corresponds to the bottom-construction of

a derivation for a judgment

$$k \rhd s$$

which means that s is a valid string with respect to counter k. More specifically, s is a valid string, given that we have already seen k left parentheses that have not yet been matched by right parentheses. We assume that $k \geq 0$ is an integer. The symbol $\rhd$ has no special meaning here—it is simply used to separate the integer k from the string s. We now develop the rules for this two-place (binary) judgment.

First, if the string s is empty then we accept if the counter is 0. This corresponds to condition (b) mentioned above.

$$\frac{}{0 \rhd \varepsilon}\ \rhd_1$$

Second, if the string s starts with an opening parenthesis, we increment the counter by 1. A less operational reading is: if s is a valid string with respect to $k+1$, then $(s$ is a valid string in stack k.

$$\frac{k+1 \rhd s}{k \rhd (s}\ \rhd_2$$

Finally, if we see a closing parenthesis at the beginning of the string, then we subtract one from the counter. It is important to check that the counter remains non-negative; otherwise we might be accepting incorrectly formed strings. A less operational reading is: if s is a valid string with counter $k > 0$ then $)s$ is a valid string with counter $k-1$.

$$\frac{k-1 \rhd s \quad (k > 0)}{k \rhd)s}\ \rhd_3$$

Since these are all the rules, the bottom-up construction of a derivation will get stuck if the string s begins with a closing parentheses and k is zero. That is, there is no rule with which we could infer $0 \rhd)s$, no matter what s is. This corresponds to condition (a) mentioned at the beginning of this discussion.

It is easy to see that this parser is inherently unambiguous. That is, when we start to construct a derivation of $0 \rhd s$ in order to parse s, then at each stage there is at most one rule that can be applied, depending on whether s is empty (rule $\rhd_1$), starts with an opening parenthesis (rule $\rhd_2$), or starts with a closing parenthesis (rule $\rhd_3$). Therefore, we can think of the judgment as describing a deterministic algorithm for parsing a string.

This judgment can be related to a *push-down automaton*. Instead of a counter k, we would have a stack (($\cdots$ (consisting of k opening parentheses. It is easy to rewrite the rules above into this form. As an aside, it turns out that every context-free grammar can be accepted by a (possibly nondeterministic) pushdown automaton, although the general construction of a pushdown automaton from a context-free grammar is more complex than in this particular example.

But does the judgment above really accept the language of properly balanced parentheses? We would like to prove that $s\ M$ if and only if $0 \rhd s$. As usual, we break this up into two separate lemmas, one for each direction.

For the first direction, we need one further lemma that captures the essence of the left-to-right processing of the input string and the use of k as a counter of unmatched open parentheses. This lemma would typically be conjectured (and then proven) only in reaction to a gap in the proof of the main theorem, but when written up it should be presented in the opposite order.

Lemma 6 (Stack)
If $k_1 \rhd s_1$ *and* $k_2 \rhd s_2$ *then* $k_1 + k_2 \rhd s_1\, s_2$

Proof: By rule induction on the derivation of $k_1 \rhd s_1$.

(Rule $\rhd_1$) Then $k_1 = 0$ and $s_1 = \varepsilon$.

$k_2 \rhd s_2$	Assumption
$k_1 + k_2 \rhd s_1\, s_2$	Since $k_1 = 0$ and $s_1 = \varepsilon$

(Rule $\rhd_2$) Then $s_1 = \texttt{(}\, s_1'$.

$k_1 + 1 \rhd s_1'$	Subderivation
$k_2 \rhd s_2$	Assumption
$k_1 + k_2 + 1 \rhd s_1'\, s_2$	By i.h.
$k_1 + k_2 \rhd \texttt{(}\, s_1'\, s_2$	By rule ($\rhd_2$)

(Rule $\rhd_3$) Then $s_1 = \texttt{)}\, s_1'$ and $k_1 > 0$.

$k_1 - 1 \rhd s_1'$	Subderivation
$k_2 \rhd s_2$	Assumption
$k_1 + k_2 - 1 \rhd s_1'\, s_2$	By i.h.
$k_1 + k_2 > 0$	Since $k_1 > 0$, $k_2 \geq 0$
$k_1 + k_2 \rhd \texttt{)}\, s_1'\, s_2$	By rule ($\rhd_3$)

■

Now we can prove the first direction of the correctness theorem for the parser.

Lemma 7
If $s\ M$ *then* $0 \rhd s$.

Proof: By rule induction on the derivation of $s\ M$.

(Rule m_1) Then $s = \varepsilon$.

$0 \rhd \varepsilon$	By rule ($\rhd_1$)

(Rule m_2) Then $s = (\ s'\ )$.

$s'\ M$	Subderivation
$0 \rhd s'$	By i.h.
$0 \rhd \varepsilon$	By rule ($\rhd_1$)
$1 \rhd)$	By rule ($\rhd_3$)
$1 \rhd s'\ )$	By Lemma 6
$0 \rhd (\ s'\ )$	By rule ($\rhd_2$)

(Rule m_3) Then $s = s_1\ s_2$.

$s_1\ M$	Subderivation
$0 \rhd s_1$	By i.h.
$s_2\ M$	Subderivation
$0 \rhd s_2$	By i.h.
$0 \rhd s_1\ s_2$	By Lemma 6

■

In order to prove the other direction (if $0 \rhd s$ then $s\ M$) we first generalize to:

If $k \rhd s$ then $\underbrace{(\ \cdots\ (}_{k}\ s\ M$.

This proof (which is left to the reader) requires another lemma, this time about the M judgment. Finally, putting the two directions together proves the correctness of our parser.

Summary. In this lecture, we introduced the concept of an *inductive definition* of a *judgment*, presented in the form of *inference rules*. As examples, we used inductive presentations of grammars and showed how to prove their equivalence via *rule induction*. We also sketched how algorithms can be presented via inference rules, using a parsing algorithm as an example. This form of presentation for algorithms, where computation is modeled by search for a deduction, is called *logic programming*,

Lecture Notes on Abstract Syntax

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 3
September 7, 2004

Grammars, as we have discussed them so far, define a formal language as a set of strings. We refer to this as the *concrete syntax* of a language. While this is necessary in the complete definition of a programming language, it is only the beginning. We further have to define at least the static semantics (via typing rules) and the dynamic semantics (via evaluation rules). Then we have to reason about their relationship to establish, for example, type soundness. Giving such definitions and proofs on strings is extremely tedious and inappropriate; instead we want to give it a more abstract form of representation. We refer to this layer of representation as the *abstract syntax* of a language. An appropriate representation vehicle are *terms* [Ch. 1].

Given this distinction, we can see that parsing is more than simply recognizing if a given string lies within the language defined by a grammar. Instead, parsing in our context should translate a string, given in concrete syntax, into an abstract syntax term. The converse problem of printing (or unparsing) is to translate an abstract syntax term into a string representation. While the grammar formalism is somewhat unwieldy when it comes to specifying the translation into abstract syntax, we see that the mechanism of judgments is quite robust and can specify both parsing and unparsing quite cleanly.

We begin by reviewing the arithmetic expression language in its concrete [Ch. 3] and abstract [Ch. 4] forms. First, the grammar in its unambigous form.[1] We implement here the decision that addition and multiplication should be left-associative (so 1+2+3 is parsed as (1+2)+3) and that

[1] We capitalize the non-terminals to avoid confusion when considering both concrete and abstract syntax in the same judgment. Also, the syntactic category of *Terms* (denoted by T) should not be confused with the terms we use to construct abstract syntax.

multiplication has precedence over addition. Such choices are somewhat arbitrary and dictated by convention rather than any scientific criteria.[2]

$$\begin{array}{lllll} \textit{Digits} & D & ::= & \texttt{0} \mid \cdots \mid \texttt{9} \\ \textit{Numbers} & N & ::= & D \mid N\,D \\ \textit{Expressions} & E & ::= & T \mid E\texttt{+}T \\ \textit{Terms} & T & ::= & F \mid T\texttt{*}F \\ \textit{Factors} & F & ::= & N \mid \texttt{(}E\texttt{)} \end{array}$$

Written in the form of five judgments.

$$\frac{}{\texttt{0 D}} \quad \cdots \quad \frac{}{\texttt{9 D}}$$

$$\frac{s\ \texttt{D}}{s\ \texttt{N}} \qquad \frac{s_1\ \texttt{N} \qquad s_2\ \texttt{D}}{s_1\ s_2\ \texttt{N}}$$

$$\frac{s\ \texttt{T}}{s\ \texttt{E}} \qquad \frac{s_1\ \texttt{E} \qquad s_2\ \texttt{T}}{s_1\texttt{+}s_2\ \texttt{E}}$$

$$\frac{s\ \texttt{F}}{s\ \texttt{T}} \qquad \frac{s_1\ \texttt{T} \qquad s_2\ \texttt{F}}{s_1\texttt{*}s_2\ \texttt{T}}$$

$$\frac{s\ \texttt{N}}{s\ \texttt{F}} \qquad \frac{s\ \texttt{E}}{\texttt{(}s\texttt{)}\ \texttt{F}}$$

The abstract syntax of the language is much simpler. It can be specified in the form of a grammar, where the universe we are working over are terms and not strings. While natural numbers can also be inductively defined in a variety of ways [Ch 1.3], we take them here as primitive mathematical objects.

$$\begin{array}{lll} \mathsf{nat} & ::= & 0 \mid 1 \mid \cdots \\ \mathsf{expr} & ::= & \mathtt{num}(\mathsf{nat}) \mid \mathtt{plus}(\mathsf{expr}, \mathsf{expr}) \mid \mathtt{times}(\mathsf{expr}, \mathsf{expr}) \end{array}$$

Presented as two judgments, we have k nat for every natural number k and the following rule for expressions

[2]The grammar given in [Ch. 3.2] is slightly different, since there addition and multiplication are assumed to be right associative.

$$\frac{k\ \mathsf{nat}}{\mathtt{num}(k)\ \mathsf{expr}}$$

$$\frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\mathtt{plus}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\mathtt{times}(t_1, t_2)\ \mathsf{expr}}$$

Now we specify the proper relation between concrete and abstract syntax through several simultaneously inductive judgments. Perhaps the easiest way to generate these judgments is to add the corresponding abstract syntax terms to each of the inference rules defining the concrete syntax.

$$\frac{}{\mathtt{0}\ \mathsf{D} \longleftrightarrow 0\ \mathsf{nat}} \quad \cdots \quad \frac{}{\mathtt{9}\ \mathsf{D} \longleftrightarrow 9\ \mathsf{nat}}$$

$$\frac{s\ \mathsf{D} \longleftrightarrow k\ \mathsf{nat}}{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}} \qquad \frac{s_1\ \mathsf{N} \longleftrightarrow k_1\ \mathsf{nat} \qquad s_2\ \mathsf{D} \longleftrightarrow k_2\ \mathsf{nat}}{s_1\ s_2\ \mathsf{N} \longleftrightarrow 10k_1 + k_2\ \mathsf{nat}}$$

$$\frac{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}}{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}} \qquad \frac{s_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{T} \longleftrightarrow t_2\ \mathsf{expr}}{s_1\mathtt{+}s_2\ \mathsf{E} \longleftrightarrow \mathtt{plus}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{s\ \mathsf{F} \longleftrightarrow t\ \mathsf{expr}}{s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}} \qquad \frac{s_1\ \mathsf{T} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{F} \longleftrightarrow t_2\ \mathsf{expr}}{s_1\mathtt{*}s_2\ \mathsf{T} \longleftrightarrow \mathtt{times}(t_1, t_2)\ \mathsf{expr}}$$

$$\frac{s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}}{s\ \mathsf{F} \longleftrightarrow \mathtt{num}(k)\ \mathsf{expr}} \qquad \frac{s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}}{\mathtt{(}s\mathtt{)}\ \mathsf{F} \longleftrightarrow t\ \mathsf{expr}}$$

When giving a specification of the form above, we should verify that the basic properties we expect, actually hold. In this case we would like to check that related strings and terms belong to the correct (concrete or abstract, respectively) syntactic classes.

Theorem 1

(i) *If* $s\ \mathsf{D} \longleftrightarrow k\ \mathsf{nat}$ *then* $s\ \mathsf{D}$ *and* $k\ \mathsf{nat}$.

(ii) *If* $s\ \mathsf{N} \longleftrightarrow k\ \mathsf{nat}$ *then* $s\ \mathsf{N}$ *and* $k\ \mathsf{nat}$.

(iii) *If* $s\ \mathsf{E} \longleftrightarrow t\ \mathsf{expr}$ *then* $s\ \mathsf{E}$ *and* $t\ \mathsf{expr}$.

(iv) *If* $s\ \mathsf{T} \longleftrightarrow t\ \mathsf{expr}$ *then* $s\ \mathsf{T}$ *and* $t\ \mathsf{expr}$.

(v) If s F $\longleftrightarrow$ t expr *then* s F *and* t expr.

Proof: Part (i) follows by cases (there are 10 cases, one for each digit).

Part (ii) follows by rule induction on the given derivation, using (i) in both cases.

Parts (iii), (iv), and (v) follow by simultaneous rule induction on the given derivation, using part (ii) in one case. Overall, there are 6 cases. In each case we can immediately appeal to the induction hypothesis on all subderivations and construct a derivation of the desired judgment from the results. ■

When implementing such a specification, we generally make a commitment as to what is considered our input and what is our output. As motivated above, parsing and unparsing (printing) are specifed by this judgment.

Definition 2 (Parsing)
Given a string s*, find a term* t *such that* s E $\longleftrightarrow$ t expr *or fail, if no such* t *exists.*

Obvious analogous definitions exist for the other syntactic categories. We can further relate parsing into abstract syntax to our definition of the syntactic categories by ascertaining that if s E then there is an abstract syntax term representing it.

Theorem 3

(i) If s D *then there is a* k *with* s D $\longleftrightarrow$ k nat.

(ii) If s N *then there is a* k *with* s N $\longleftrightarrow$ k nat.

(iii) If s E *then there is a* t *with* s E $\longleftrightarrow$ t expr.

(iv) If s T *then there is a* t *with* s T $\longleftrightarrow$ t expr.

(v) If s F *then there is a* t *with* s F $\longleftrightarrow$ t expr.

Proof: By cases or straightforward rule induction as in the proof of Theorem 1. ■

Now we can refine our notion of ambiguity to take into account the abstract syntax that is constructed. This is slightly more relaxed that requiring the uniqueness of derivations, because different derivations could still lead to the same abstract syntax term.

Definition 4 (Ambiguity of Parsing)
A parsing problem is ambiguous if for a given string s there exist two distinct terms t_1 and t_2 such that s E $\longleftrightarrow t_1$ expr and s E $\longleftrightarrow t_2$ expr.

Unparsing is just the reverse of parsing: we are given a term t and have to find a concrete syntax representation for it. Unparsing is usually total (every term can be unparsed) and inherently ambiguous (the same term can be written as several strings). An example of this ambiguity is the insertion of additional redundant parentheses. Therefore, any unparser must use heuristics to choose among different alternative string representations.

Definition 5 (Unparsing)
Given a term t such that t expr, find a string s such that s E $\longleftrightarrow t$ expr.

The ability to use judgments as the basis for implementation of different tasks is evidence for their flexibility. Often, it is not difficult to "translate" a judgment into an implementation in a high-level language such as ML, although in some cases it might require significant ingenuity and some advanced techniques.

Our little language of arithmetic expressions serves to illustrate various ideas, such as the distinction between concrete syntax and abstract syntax, but it is too simple to exhibit various other phenomena and concepts. One of the most important one is that of a variable, and the notion of variable binding and scope. In order to discuss variables in isolation, we extend our language by a new form of expression to name preliminary results. For example,

$$\texttt{let}\ x\ \texttt{=2*3 in}\ x\texttt{+}x\ \texttt{end}$$

should evaluate to 12, but only compute the value of `2*3` once.

First, the concrete syntax, showing only the changed or new cases.

$$\begin{array}{llll} \textit{Variables} & X & ::= & \textit{(any identifier)} \\ \textit{Factors} & F & ::= & N \mid \texttt{(}E\texttt{)} \mid \texttt{let}\ X\ \texttt{=}\ E\ \texttt{in}\ E\ \texttt{end} \mid X \end{array}$$

We ignore here the question what constitutes a legal identifier. Presumably it should avoid keywords (such as `let`, `b`), special symbols, such as `+`, and be surrounded by whitespace. In an actual language implementation a *lexer* breaks the input string into keywords, special symbols, numbers, and identifiers that are the processed by the parser.

The first approach to the abstract syntax would be to simply introduce a new abstract syntactic category of *variable* [Ch. 5.1] and a new operator `let` with three arguments, $\texttt{let}(x, e_1, e_2)$, where x is a variable and e_1 and e_2 are

terms representing expressions. Furthermore, we allow an occurrence of a variable x as a term. However, this approach does not clarify which occurrences of a variable are *binding occurrences*, and to which binder a variable occurrence refers. For example, to see that

$$\texttt{let}\ x = \texttt{1}\ \texttt{in}\ \texttt{let}\ x = x\texttt{+1}\ \texttt{in}\ x\texttt{+}x\ \texttt{end}\ \texttt{end}$$

evaluates to 4, we need to know which occurrences of x refer to which values. Rules for scope resolution [Ch. 5.1] dictate that it should be interpreted the same as

$$\texttt{let}\ x_1 = \texttt{1}\ \texttt{in}\ \texttt{let}\ x_2 = x_1\texttt{+1}\ \texttt{in}\ x_2\texttt{+}x_2\ \texttt{end}\ \texttt{end}$$

where there is no longer any potential ambiguity. That is, the scope of the variable x in

$$\texttt{let}\ x = s_1\ \texttt{in}\ s_2\ \texttt{end}$$

is s_2 but not s_1.

A uniform technique to encode the information about the scope of variables is called *higher-order abstract syntax* [Ch. 5]. We add to our language of terms a construct $x.t$ which binds x in the term t. Every occurrence of x in t that is not shadowed by another binding $x.t'$, refers to the shown top-level abstraction. Such variables are a new primitive concept, and, in particular, a variable can be used as a term (in addition to the usual operator-based terms). We would extend our judgment relating concrete and abstract syntax by

$$\frac{x\ \mathsf{X} \qquad s_1\ \mathsf{E} \longleftrightarrow t_1\ \mathsf{expr} \qquad s_2\ \mathsf{E} \longleftrightarrow t_2\ \mathsf{expr}}{\texttt{let}\ x = s_1\ \texttt{in}\ s_2\ \texttt{end} \longleftrightarrow \texttt{let}(t_1, x.t_2)\ \mathsf{expr}} \qquad \frac{x\ \mathsf{X}}{x\ \mathsf{E} \longleftrightarrow x\ \mathsf{expr}}$$

and allow for expressions

$$\frac{}{\overline{x}\ \mathsf{expr}} \qquad \frac{t_1\ \mathsf{expr} \qquad t_2\ \mathsf{expr}}{\texttt{let}(t_1, x.t_2)\ \mathsf{expr}}$$

Note that we translate an identifier x to an identically named variable x in higher-order abstract syntax. Moreover, we view variables in higher-order abstract syntax as a new kind of term, so we do not check explicitly the x's are in fact variables—it is implied that they are.

We emphasize that the laws for scope resolution of `let`-expressions are directly encoded in the higher-order abstract representation. We investigate the laws underlying such representations in Lecture 4 [Ch. 5.3].

We can formulate the language of abstract syntax for arithmetic expressions in a more compact notation as a grammar.

$$\begin{array}{lll} \mathsf{nat} & ::= & 0 \mid 1 \mid \cdots \\ \mathsf{expr} & ::= & \mathtt{num}(\mathsf{nat}) \mid \mathtt{plus}(\mathsf{expr}, \mathsf{expr}) \mid \mathtt{times}(\mathsf{expr}, \mathsf{expr}) \\ & & \mid x \mid \mathtt{let}(\mathsf{expr}, x.\mathsf{expr}) \end{array}$$

As a concrete example, consider the string

$$\mathtt{let}\ x_1 = \mathtt{1}\ \mathtt{in}\ \mathtt{let}\ x_2 = x_1 \mathtt{+} \mathtt{1}\ \mathtt{in}\ x_2 \mathtt{+} x_2\ \mathtt{end}\ \mathtt{end}$$

which, in abstract syntax, would be represented as

$$\mathtt{let}(\mathtt{num}(1), x_1.\mathtt{let}(\mathtt{plus}(x_1, \mathtt{num}(1)), x_2.\mathtt{plus}(x_2, x_2)))$$

Lecture Notes on Static and Dynamic Semantics

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 4
September 9, 2004

In this lecture we illustrate the basic concepts underlying the static and dynamic semantics of a programming language on a very simple example: the language of arithmetic expression augmented by variables and definitions.

The static and dynamic semantics are properties of the abstract syntax (terms) rather than the concrete syntax (strings). Therefore we will deal exclusively with abstract syntax here.

The static semantics can further be decomposed into two parts: variable scope and rules of typing. They determine how to interpret variables, and discern the meaningful expressions. As we saw in the last lecture, variable scope is encoded directly into the terms representing the abstract syntax. In this lecture we further discuss the laws governing variable binding on terms. The second step will be to give the rules of typing in the form of an inductively defined judgment. This is not very interesting for arithmetic expressions, comprising only a single type, but it serves to illustrate the ideas.

The dynamic semantics varies more greatly between different languages and different levels of abstraction. We will only give a very brief introduction here and continue the topic in the next lecture.

The basic principle of variable binding called *lexical scoping* is that the name of a bound variable should not matter. In other words, consistently renaming a variable in a program should not affect its meaning. Everything below will follow from this principle.

We now make this idea of "consistent renaming of variables" more precise. The development in [Ch. 5.3] takes simultaneous substitution as a

primitive; we avoid the rather heavy notation by only dealing with a single substitution at a time. This goes hand in hand with the decision that binding prefixes such as $x.t$ only ever bind a single variable, and not multiple ones. We use the notation $\{y/x\}t$ to denote the result of substituting y for x in t, yet to be defined. With that we will define renaming of x to y with the equation

$$x.t =_\alpha y.\{y/x\}t$$

which can be applied multiple times, anywhere in a term. For this to preserve the meaning, y most not already occur free in $x.t$, because otherwise the free occurrence of y would be *captured* by the new binder.

As an example, consider the term

$$\mathtt{let}(\mathtt{num}(1), x.\mathtt{let}(\mathtt{plus}(x, \mathtt{num}(1)), y.\mathtt{plus}(y, x)))$$

which should evaluate to $\mathtt{num}(3)$. It should be clear that renaming y to x should be disallowed. The resulting term

$$\mathtt{let}(\mathtt{num}(1), x.\mathtt{let}(\mathtt{plus}(x, \mathtt{num}(1)), x.\mathtt{plus}(x, x)))$$

means something entirely different and would evaluate to $\mathtt{num}(4)$.

To make this side condition more formal, we define the set of free variables in a term.

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{\, x \,\} \\ \mathrm{FV}(o(t_1, \ldots, t_n)) & = & \bigcup_{1 \le i \le n} \mathrm{FV}(t_i) \\ \mathrm{FV}(x.t) & = & \mathrm{FV}(t) \setminus \{\, x \,\} \end{array}$$

So before defining the substitution $\{y/x\}t$ we restate the rule defining variable renaming, also called α-conversion, with the proper side condition:

$$x.t =_\alpha y.\{y/x\}t \qquad \text{provided } y \notin \mathrm{FV}(t)$$

Now back to the definition of substitution of one variable y for another variable x in a term t, $\{y/x\}t$. The definition recurses over the structure of a term.[1]

$$\begin{array}{rcll} \{y/x\}x & = & y & \\ \{y/x\}z & = & z & \text{provided } x \neq z \\ \{y/x\}o(t_1, \ldots, t_n) & = & o(\{y/x\}t_1, \ldots, \{y/x\}t_n) & \\ \{y/x\}x.t & = & x.t & \\ \{y/x\}z.t & = & x.\{y/x\}t & \text{provided } x \neq z \text{ and } y \neq z \\ \{y/x\}y.t & & \text{undefined} & \text{provided } x \neq y \end{array}$$

[1] It can in fact be seen as yet another form of inductive definition, but we will not formalize this here.

Note that substitution is a *partial* operation. The reason the last case must be undefined is because any occurrence of x in t would be replaced by y and thereby captured. As an example while this must be ruled out, reconsider

$$\mathtt{let}(\mathtt{num}(1), x.\mathtt{let}(\mathtt{plus}(x, \mathtt{num}(1)), y.\mathtt{plus}(y, x)))$$

which evaluates to $\mathtt{num}(3)$. If we were allowed to rename x to y we would obtain

$$\mathtt{let}(\mathtt{num}(1), y.\mathtt{let}(\mathtt{plus}(y, \mathtt{num}(1)), y.\mathtt{plus}(y, y)))$$

which once again means something entirely different and would evaluate to $\mathtt{num}(4)$.

In the operational semantics we need a more general substitution, because we need to substitute one term for a variable in another term. We generalize the definition above, taking care to rewrite the side condition on substitution in a slightly more general, but consistent form, in order to prohibit variable capture.

$$\begin{array}{rcll}
\{u/x\}x & = & u & \\
\{u/x\}z & = & z & \text{provided } x \neq z \\
\{u/x\}o(t_1, \ldots, t_n) & = & o(\{u/x\}t_1, \ldots, \{u/x\}t_n) & \\
\{u/x\}x.t & = & x.t & \\
\{u/x\}z.t & = & z.\{u/x\}t & \text{provided } x \neq z \text{ and } z \notin \mathrm{FV}(u) \\
\{u/x\}z.t & & \text{undefined} & \text{provided } x \neq z \text{ and } z \in \mathrm{FV}(u)
\end{array}$$

In practice we would like to treat substitution as a total operation. This cannot be justified on terms, but, surprisingly, it works on α-equivalence classes of terms! Since we want to identify terms that only differ in the names of their bound variables, this is sufficient for all purposes in the theory of programming languages. More formally, the following theorem (which we will not prove) justifies treating substitution as a total operation.

Theorem 1 (Substitution and α-Conversion)

(i) If $u =_\alpha u'$, $t =_\alpha t'$, and $\{u/x\}t$ and $\{u'/x\}t'$ are both defined, then $\{u/x\}t =_\alpha \{u'/x\}t'$.

(ii) Given u, x, and t, then there always exists a $t' =_\alpha t$ such that $\{u/x\}t'$ is defined.

We sketch the proof of part *(ii)*, which proceeds by induction on the size of t. If $\{u/x\}t$ is defined we choose t' to be t. Otherwise, then somewhere the last clause in the definition of substitution applies and there is a binder

$z.t_1$ in t such that $z \in \mathrm{FV}(u)$. Then we can rename z to a new variable z' which occurs neither in free in u nor free in $z.t_1$ to obtain $z'.t'_1$. Now we can continue with $z'.\{u/x\}t'_1$. by an appeal to the induction hypothesis.

The algorithm described in this proof is in fact the definition of *capture-avoiding substitution* which makes sense whenever we are working modulo α-equivalence classes of terms. Fortunately, this will always be the case for the remainder of this course.

With the variable binding, renaming, and substitution understood, we can now formulate a first version of the typing rules for this language. Because there is only one type, `nat`, the rules are somewhat trivialized. Their only purpose for this small language is to verify that an expression e is *closed*, that is, $\mathrm{FV}(e) = \{\,\}$.

A first judgmental way to express this would be the following[2]:

$$\frac{k\ \mathsf{nat}}{\mathtt{num}(k) : \mathtt{nat}}$$

$$\frac{e_1 : \mathtt{nat} \quad e_2 : \mathtt{nat}}{\mathtt{plus}(e_1, e_2) : \mathtt{nat}} \qquad \frac{e_1 : \mathtt{nat} \quad e_2 : \mathtt{nat}}{\mathtt{times}(e_1, e_2) : \mathtt{nat}}$$

$$\frac{e_1 : \mathtt{nat} \quad \{e_1/x\}e_2 : \mathtt{nat}}{\mathtt{let}(e_1, x.e_2) : \mathtt{nat}}$$

While this is perfectly correct, it has the potential problem that it re-checks e_1 for every occurrence of x in e_2. This could be avoided by substituting a fixed value such as $\mathtt{num}(0)$ for x and checking the result.

A more common (and more scalable) alternative is to use a new judgment form, a so-called *hypothetical judgment*. We write it as

$$J_1, \ldots, J_n \vdash J$$

which means that J follows from assumptions $J_1, \ldots, J_n$. Its most basic property is that

$$J_1, \ldots, J_i, \ldots J_n \vdash J_i$$

always holds, which should be obvious: if an assumption is identical to the judgment we are trying to derive, we are done. We will nonetheless restate instances of this general principle for each case.

The particular form of hypothetical judgment we consider is

$$x_1{:}\mathtt{nat}, \ldots, x_n{:}\mathtt{nat} \vdash e : \mathtt{nat}$$

which should be read:

[2] suggested by a student in class

> *Under the assumption that variables $x_1, \ldots, x_n$ stand for natural numbers, e has the type of natural number.*

We usually abbreviate a whole sequence of assumptions with the letter Γ.[3] We write '$\cdot$' for an empty collection of assumptions, and we abbreviate $\cdot, x{:}\mathtt{nat}$ by $x{:}\mathtt{nat}$. In order to avoid ambiguities, we always assume that all variables declared in a context are distinct.

The typing judgment is defined by the following rules.

$$\frac{x{:}\mathtt{nat} \in \Gamma}{\Gamma \vdash x : \mathtt{nat}} \qquad \frac{k\ \mathsf{nat}}{\Gamma \vdash \mathtt{num}(k) : \mathtt{nat}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{plus}(e_1, e_2) : \mathtt{nat}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{times}(e_1, e_2) : \mathtt{nat}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{nat} \quad \Gamma, x{:}\mathtt{nat} \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : \mathtt{nat}}$$

In the last rule some care has to be taken to make sure that x is not declared twice in the context. If the variable x bound in $\mathtt{let}(e_1, x.e_2)$ is already declared, we use the assumption that we work modulo α-equivalence classes and rename the variable x to a fresh variable x' before applying the rule.

The point of being interested in typing for this small language is only to guarantee that there are no free variables in a term to the evaluation will not get stuck. This property can easily be verified.

Theorem 2
If $\cdot \vdash e : \mathtt{nat}$ *then* $\mathrm{FV}(e) = \{\}$.

Proof: We cannot prove this directly by rule induction, since the second premise of the rule for `let` introduces an assumption. So we generalizing to

> *If* $x_1{:}\mathtt{nat}, \ldots, x_n{:}\mathtt{nat} \vdash e : \mathtt{nat}$ *then* $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$.

This generalized statement can be proved easily by rule rule induction. ■

Next we would like to give the operational semantics, specifying the value of an expression. We represent values also as expressions, although they are restricted to have the form $\mathtt{num}(k)$. Generally, when we write an

[3] In [Ch. 6] this is written instead as $\Gamma \vdash e$ ok, where Γ is a set of variables. Since there is only one type, the two formulations are clearly equivalent.

expression as v we imply that it is a value and therefore has the form $\mathsf{num}(k)$ for some k.

There are multiple ways to specify the operational semantics, for example as a structured operational semantics [Ch. 7.1] or as an evaluation semantics [Ch. 7.2]. We give two forms of evaluation semantics here, which directly relate an expression to its value.

The first way employs a hypothetical judgment in which we make assumptions about the values of variables. It is written as

$$x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n \vdash e \Downarrow v.$$

We call $x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n$ an *environment* and denote an environment by η. It is important that all variables x_i in an environment are distinct so that the value of a variable is uniquely determined.

$$\frac{x \Downarrow v \in \eta}{\eta \vdash x \Downarrow v} \qquad \frac{}{\eta \vdash \mathsf{num}(k) \Downarrow \mathsf{num}(k)}$$

$$\frac{\eta \vdash e_1 \Downarrow \mathsf{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathsf{num}(k_2)}{\eta \vdash \mathtt{plus}(e_1, e_2) \Downarrow \mathsf{num}(k_1 + k_2)} \qquad \frac{\eta \vdash e_1 \Downarrow \mathsf{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathsf{num}(k_2)}{\eta \vdash \mathtt{times}(e_1, e_2) \Downarrow \mathsf{num}(k_1 \times k_2)}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2} \ (x \text{ not declared in } \eta)$$

In the rule for `let` we make the assumption that the value of x is v_1 while evaluating e_2. One may be concerned that this operational semantics is partial, in case bound variables with the same name occur nested in a term. However, since we working with α-equivalences classes of terms we can always rename the inner bound variable to that the rule for `let` applies. We will henceforth not make such a side condition explicit, using the general convention that we rename bound variables as necessary so that contexts or environment declare only distinct variables.

An alternative semantics uses substitution instead of environments. For this judgment we evaluate only closed terms, so no hypothetical judgment is needed.

$$\textit{No rule for variables } x \qquad \frac{}{\mathsf{num}(k) \Downarrow \mathsf{num}(k)}$$

$$\frac{e_1 \Downarrow \mathsf{num}(k_1) \quad e_2 \Downarrow \mathsf{num}(k_2)}{\mathtt{plus}(e_1, e_2) \Downarrow \mathsf{num}(k_1 + k_2)} \qquad \frac{e_1 \Downarrow \mathsf{num}(k_1) \quad e_2 \Downarrow \mathsf{num}(k_2)}{\mathtt{times}(e_1, e_2) \Downarrow \mathsf{num}(k_1 \times k_2)}$$

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\mathtt{let}(e_1, x.e_2) \Downarrow v_2}$$

We postpone a discussion on the relationship between the two forms of semantics, but we will considered how typing is related to the operational semantics. Clearly, we cannot pick rules arbitrarily, but typing must reflect the operational behavior of programs and, conversely, the operational semantics must reflect typing. As first example of this relationship we show that well-typed and closed arithmetic expressions always evaluate to a value.

If $\cdot \vdash e : \texttt{nat}$ *then* $\cdot \vdash e \Downarrow \texttt{num}(k)$ *for some* k*.*

We cannot prove this directly by induction, since the second premise in the case of the typing rule for $\texttt{let}(e_1, x.e_2)$ would have the form $\cdot, x{:}\texttt{nat} \vdash e_2 : \texttt{nat}$. This does not match the induction hypothesis where the context is required to be empty.

If we look at the rules for typing and evaluation side-by-side, we see that if we start with an empty context and environment, the set of variables in the derivations always correspond. We define that η *matches* Γ if η defines values for the same variables as are declared in Γ.

Lemma 3 (Evaluation in Environment)
If $\Gamma \vdash e : \texttt{nat}$ *and* η *matches* Γ *then* $\eta \vdash e \Downarrow \texttt{num}(k)$ *for some* k*.*

Proof: By rule induction on the derivation of $\Gamma \vdash e : \texttt{nat}$.

(Rule for $\texttt{num}(k)$) Then

$\eta \vdash \texttt{num}(k) \Downarrow \texttt{num}(k)$	By rule

(Rule for $\texttt{plus}(e_1, e_2)$) Then

$\Gamma \vdash e_1 : \texttt{nat}$	Subderivation
$\Gamma \vdash e_2 : \texttt{nat}$	Subderivation
η matches Γ	Assumption
$\eta \vdash e_1 \Downarrow \texttt{num}(k_1)$ for some k_1	By i.h.
$\eta \vdash e_2 \Downarrow \texttt{num}(k_2)$ for some k_2	By i.h.
$\eta \vdash \texttt{plus}(e_1, e_2) \Downarrow \texttt{num}(k_1 + k_2)$	By rule

(Rule for $\texttt{times}(e_1, e_2)$) Analogous to previous case.

(Rule for variable x) Then

x:nat $\in \Gamma$	Subderivation
η matches Γ	Assumption
$x \Downarrow \mathsf{num}(k)$ for some k	By defn. of matching
$\eta \vdash x \Downarrow \mathsf{num}(k)$	By rule

(Rule for $\mathtt{let}(e_1, x.e_2)$) Then

$\Gamma \vdash e_1 :$ nat	Subderivation
η matches Γ	Assumption
$\eta \vdash e_1 \Downarrow \mathsf{num}(k)$	By i.h.
$\eta, x \Downarrow \mathsf{num}(k)$ matches Γ, x:nat	By defn. of matching
Γ, x:nat $\vdash e_2 :$ nat	Subderivation
$\eta, x \Downarrow \mathsf{num}(k) \vdash e_2 \Downarrow \mathsf{num}(k_2)$	By i.h.
$\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow \mathsf{num}(k_2)$	By rule

■

Lecture Notes on
A Functional Language

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 5
September 14, 2004

We now introduce MinML, a small fragment of ML that serves to illustrates key points in its design and key techniques for verifying its properties. The treatment here is somewhat cursory; see [Ch. 9] for additional material. Roughly speaking, MinML arises from the arithmetic expression language by adding booleans, functions, and recursion. Functions are (almost) first-class in the sense that they can occur anywhere in an expression, rather than just at the top-level as in other languages such as C. This has profound consequences for the required implementation techniques (to which we will return later), but it does not affect typing in an essential way.

First, we give the grammar for the higher-order abstract syntax. For the concrete syntax, please refer to Assignment 2.

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{arrow}(\tau_1, \tau_2) \\
\text{Integers} & n & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
\text{Primops} & o & ::= & \texttt{plus} \mid \texttt{minus} \mid \texttt{times} \mid \texttt{negate} \\
 & & \mid & \texttt{equals} \mid \texttt{lessthan} \\
\text{Expressions} & e & ::= & \texttt{num}(n) \mid o(e_1, \ldots, e_n) \\
 & & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}(e, e_1, e_2) \\
 & & \mid & \texttt{let}(e_1, x.e_2) \\
 & & \mid & \texttt{fn}(\tau, x.e) \mid \texttt{apply}(e_1, e_2) \\
 & & \mid & \texttt{rec}(\tau, x.e) \\
 & & \mid & x
\end{array}
$$

Our typing judgment that sorts out the well-formed expressions has the form $\Gamma \vdash e : \tau$, where a context Γ has the form $\cdot, x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. It is a hy-

pothetical judgment as explained in the previous lecture. Our assumption that all variables x_i declared in a context must be distinct is still in force, which means that the rule

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{VarTyp}$$

is unambiguous since there can be at most one declaration for x in Γ.

We have already discussed arithmetic expressions; booleans constitute a similar basic type. Unlike languages such as C, integers and booleans are strictly separate types, avoiding some common confusions and errors. Below are the typing rules related to booleans.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{equals}(e_1, e_2) : \texttt{bool}}\ \textit{EqualsTyp}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp}$$

Perhaps the only noteworthy point here is that the two branches of a conditional must have the same type. This is because we cannot know at type-checking time which branch will be taken at run-time. We are therefore conservative, asserting only that the result of the conditional will definitely have type τ if each branch has type τ. Later in this class, we will see a type system that can more accurately analyze conditionals so that, for example, `if true then 1 else false` could be given a type (which is impossible here).

A more important extension from our first language of arithmetic expressions is the addition of functions. In mathematics we are used to describe functions in the form $f(x) = e$, for example $f(x) = x^2 + 1$. In a functional language we want a notation for the function f itself. The abstract (mathematical) notation for this concept is λ-abstraction, written $f = \lambda x.e$. The above example would be written as $f = \lambda x.x^2 + 1$.

In the concrete syntax of MinML we express $\lambda x{:}\tau.e$ as `fn x:t => e`; in our abstract syntax it is written as $\texttt{fn}(\tau, x.e)$. This is an illustration of the unfortunate situation that we generally have to deal with at least three ways of expressing the same concepts. One is the mathematical notation, one is the concrete syntax, and one is the abstract syntax. In research papers, one mostly uses mathematical notation or pseudo-concrete syntax

that really stands for abstract syntax but is easier to read. Inevitably, we will also start sliding between levels of discourse which is acceptable as long as we always know what we *really* mean.

Returning to functions, the typing rules are rather straightforward.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}(\tau_1, x.e) : \texttt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

$$\frac{\Gamma \vdash e_1 : \texttt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

Keep in mind that in the rule *FnTyp*, the variable x must not already be declared in Γ. We can always rename x in $\texttt{fn}(\tau, x.e)$ to satisfy this condition, because we treat abstract syntax as α-equivalence classes, that is, modulo variable renaming.

Functions defined with the language given so far are rather limited. For example, there is no way to define the exponential function from multiplication and addition, because there is no way to express recursion implicit in the definition

$$\begin{array}{lcl} 2^0 & = & 1 \\ 2^n & = & 2 \times 2^{n-1} \quad \text{for } n > 0. \end{array}$$

We address this problem in the next lecture when we introduce the concept of recursion.

Below is a summary of the typing rules for the language. We show only the case of one operator—the others are analogous.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{VarTyp} \qquad \frac{}{\Gamma \vdash \texttt{num}(n) : \texttt{int}}\ \textit{NumTyp}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{equals}(e_1, e_2) : \texttt{bool}}\ \textit{EqualsTyp}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}(e, e_1, e_2) : \tau}\ \textit{IfTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}(e_1, x.e_2) : \tau_2}\ \textit{LetTyp}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) : \mathtt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

We specify the operational semantics as a *natural semantics* also called a *big-step semantics*. As the semantics for our small language of expressions, it relates an expression to its final value (if it has one), but it does not directly specify each step of evaluation. We use substitution instead of environments for simplicity, so the judgment has the form $e \Downarrow v$ where we assume that $\cdot \vdash e : \tau$. We also define the judgment e value which expresses that e is a value (written v).

Integers This is quite simple and as for arithmetic expressions. First, only numbers are values.

$$\frac{}{\mathtt{num}(k)\ \mathsf{value}}$$

Then the rules for evaluations; we only show the rules for the primitive equality operator.

$$\frac{}{\mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2) \quad (k_1 = k_2)}{\mathtt{equals}(e_1, e_2) \Downarrow \mathtt{true}}$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2) \quad (k_1 \neq k_2)}{\mathtt{equals}(e_1, e_2) \Downarrow \mathtt{false}}$$

Booleans First, `true` and `false` are values.

$$\frac{}{\mathtt{true}\ \mathsf{value}} \qquad \frac{}{\mathtt{false}\ \mathsf{value}}$$

Then, the decision on which branch of a conditional to evaluate is based on the return value of the condition.

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}}$$

$$\frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v_1}{\texttt{if}(e, e_1, e_2) \Downarrow v_1} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v_2}{\texttt{if}(e, e_1, e_2) \Downarrow v_2}$$

Definitions This remains unchanged from the arithmetic expression language.

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\texttt{let}(e_1, x.e_2) \Downarrow v_2}$$

Functions It is often claimed that functions are "first-class", but this is not quite true, since we cannot observe the structure of functions in the same way we can observe booleans or integers. Therefore, there is no need to evaluate the body of a function, and in fact we could not since it is not closed and we would get stuck when encountering the function parameter. So, any function by itself is a value.

$$\frac{}{\texttt{fn}(\tau, x.e)\ \mathsf{value}}$$

The second question is if we have to evaluate the argument in a function call before performing the call. Both answers are sensible. In languages like C, Java, or ML, function arguments are evaluated. This also corresponds to mathematical practice. For example, a function from integers to integers takes integers as arguments, not expressions. Of course, we may perform reasoning to deduce equations involving functions, but this is quite distinct from computation. In other languages like Haskell, function arguments are not evaluated. We will discuss this possibility and its applications in more detail later in this course. Given that we evaluate arguments, we call our language *call-by-value* and define it by the following rules.

$$\frac{}{\texttt{fn}(\tau, x.e) \Downarrow \texttt{fn}(\tau, x.e)}$$

$$\frac{e_1 \Downarrow \texttt{fn}(\tau_2, x.e_1') \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_1' \Downarrow v}{\texttt{apply}(e_1, e_2) \Downarrow v}$$

Which theorems regarding the operational semantics make sense in this setting? First, we can state that evaluation, if it terminates, should always result in a value. Second, we can state that evaluation preserves the type of the expression all the way to its value. Finally, we want to claim that the language is deterministic, that is, the value of an expression (if it exists) is uniquely determined.

1. (Evaluation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then v value.
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$, then $\cdot \vdash v : \tau$
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \Downarrow v'$ and $e \Downarrow v''$ then $v' = v''$.

We will return to the problem of proving these and similar theorems in the next lecture. Note that this does not exhaust the possibilities of possible theorems, and there are many other properties which may be of interest for specific purposes in the implementation or use of a language.

We conclude the lecture with some discussion on how these inference rules may be viewed as specifications of algorithms. Interestingly, the evaluation judgment can be viewed as an algorithm for evaluating an expression, and the typing judgment can be viewed as an algorithm for typechecking. However, not every judgment can be interpreted in this way, so we must take some care to ensure this kind of reading is meaningful. The kind of reasoning we apply here is also the kind of reasoning required to turn the judgments and rules into functional implementations (say, using ML or Haskell as an implementation language). This sort of analysis is routine for programming language researchers, but it is rarely made explicit.

We begin with the evaluation judgment. We would like to read the rules for evaluation as an algorithm for computing the value of an expression. So we commit to saying that in the judgment $e \Downarrow v$, e is the input (given) and v is the output (to be computed). Now we analyze each rule to see if we can see how to compute v given e.

Integers For integers, the analysis is entirely straightforward.

$$\frac{}{\mathsf{num}(k) \Downarrow \mathsf{num}(k)}$$

Given the input $\texttt{num}(k)$ we can indeed compute the (identical) output $\texttt{num}(k)$.

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 = k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{true}}$$

$$\frac{e_1 \Downarrow \texttt{num}(k_1) \quad e_2 \Downarrow \texttt{num}(k_2) \quad (k_1 \neq k_2)}{\texttt{equals}(e_1, e_2) \Downarrow \texttt{false}}$$

Given the input $\texttt{equals}(e_1, e_2)$ we know both e_1 and e_2. Since e_1 is known, by induction hypothesis[1] we can compute k_1. From the second premise we can obtain k_2. Then we can compare these values and return either `true` or `false`, depending on which rule applies.

We skip Booleans and definitions, and go right to the most complicated case of functions.

Functions Function expression evaluate to themselves, so if we know the input we can return the output.

$$\frac{}{\texttt{fn}(\tau, x.e) \Downarrow \texttt{fn}(\tau, x.e)}$$

$$\frac{e_1 \Downarrow \texttt{fn}(\tau_2, x.e_1') \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_1' \Downarrow v}{\texttt{apply}(e_1, e_2) \Downarrow v}$$

For function application, the reasoning is more complex.

We are given $\texttt{apply}(e_1, e_2)$.
Hence we know e_1 and e_2.
By i.h. we know $\texttt{fn}(\tau_2, x.e_1')$.
By i.h. we know v_2.
We therefore can calculate $\{v_2/x\}e_1'$
By i.h. we can compute v
Therefore we can return v

For the typing judgment, we can perform a similar analysis. But first we have to decide what are the inputs, and what are the outputs of the judgment $\Gamma \vdash e : \tau$. We might try[2] to use both Γ, e, and τ as inputs and decide

[1] This reasoning could be formally set up as an induction, showing that if e is given then v can be computed (assuming it exists at all). Even though we do not formalize this, we still refer to the "induction hypothesis" when analyzing the premises.

[2] suggested in lecture by a student

if the judgments holds or not (that is, either succeed or fail). Unfortunately, this does not work for function application $\texttt{apply}(e_1, e_2)$: we cannot determine the type of the argument e_2.

$$\frac{\Gamma \vdash e_1 : \texttt{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau} \; \textit{AppTyp}$$

We assume that Γ, $\texttt{apply}(e_1, e_2)$ and τ are known. But we cannot apply the induction hypothesis to the first premise, because τ_2 is unknown. Similarly, we cannot apply the induction hypothesis to the second premise, since τ_2 is unknown. We are therefore stuck, which means that we cannot easily interpret the typing rules for checking a given expression against a given type.

Fortuntely, we can assume Γ and e as inputs and generate τ as output, or fail (if the expression is not well-typed). In that case we analyze the rule as follows.

Γ, $\texttt{apply}(e_1, e_2)$ are given.
Therefore, e_1 and e_2 are known.
By i.h. a τ_1 such that $\Gamma \vdash e_1 : \tau_1$ can be computed (or we fail).
By i.h. τ_2 can be computed from the second premise (or we fail).
Now we check if $\tau_1 = \texttt{arrow}(\tau_2, \tau)$ for some τ.
If no, we fail.
If yes, we return τ.

Finally, we consider functional abstraction.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}(\tau_1, x.e) : \texttt{arrow}(\tau_1, \tau_2)} \; \textit{FnTyp}$$

Γ, $\texttt{fn}(\tau_1, x.e)$ are given.
Hence τ_1, x, and e are known.
By i.h. τ_2 can be computed (or we may fail).
If we succeed, we can construct $\texttt{arrow}(\tau_2, \tau_2)$.

This reasoning required that the type τ_1 be present in the expression, otherwise we could not apply the induction hypothesis. This is precisely the reason why τ_1 in in fact required in the syntax. ML does not require this type, because it performs a much more complicated analysis of expressions called *type inference*. Briefly, does not compute exact types but creates placeholders and generates a potentially large set of equational constraints

between types and placeholders which must be satisfied for the expression to be well-typed. It then solves these constraints by an algorithm that resembles Gaussian elimination for solving linear arithmetic equalities. We will come back to this process in a later lecture.

Lectures Notes on Type Safety

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 6
September 16, 2004

Before we discuss type safety, we introduce recursion into the language. Without recursion, the set of functions that can be defined on natural numbers is of course very limited. Rather than tying recursion to functions, the way it is done in [Ch. 9], we introduce it here as a separate concept. In concrete syntax, we write `rec x:t => e` for a recursive expression. The same is represented in abstract syntax as $\mathtt{rec}(\tau, x.e)$, which makes explicit that x is a bound variable with scope e. The intuitive meaning of $\mathtt{rec}(\tau, x.e)$ is that it should be "equal" to its unfolding, that is, the result of substituting the whole expressions for x in e. That is, in some sense we would like to equate $\mathtt{rec}(\tau, x.e)$ with $\{\mathtt{rec}(\tau, x.e)/x\}e$. In the operational semantics, this is manifest in the rule

$$\frac{\{\mathtt{rec}(\tau, x.e)/x\}e \Downarrow v}{\mathtt{rec}(\tau, x.e) \Downarrow v}$$

As an example, consider the exponential function

$$\begin{array}{lcl} 2^0 & = & 1 \\ 2^n & = & 2 \times 2^{n-1} \quad \text{for } n > 0. \end{array}$$

In order to express this with the recursion construction, we write

$$\begin{array}{l} \mathtt{rec}(\mathtt{arrow}(\mathtt{int}, \mathtt{int}), p.\mathtt{fn}(\mathtt{int}, n. \\ \quad \mathtt{if}(\mathtt{equals}(n, \mathtt{num}(0)), \\ \qquad \mathtt{num}(1), \\ \qquad \mathtt{times}(\mathtt{num}(2), \mathtt{apply}(p, \mathtt{minus}(x, \mathtt{num}(1)))))). \end{array}$$

or, in concrete syntax:

```
rec p:int -> int => fn n:int =>
  if n = 0
     then 1
     else 2 * p (x - 1)
```

You should convince yourself on the example above that unfolding yields the correct behavior. As for the typing rule: the whole expression must have the same type as x, so that the substitution $\{\texttt{rec}(\tau, x.e)/x\}$ makes sense. The same type τ must also be the type of e, because the value of e is returned as the value of the recursive expression.

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}(\tau, x.e) : \tau}\ \textit{RecTyp}$$

As in function expressions, the type τ is recorded in the syntax so that type-checking can be implemented in a simple manner.

In MinML, most useful recursions have the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e)),$$

because most other recursive expressions will not terminate (try, for example, $\texttt{rec}(\texttt{int}, x.x)$). We therefore introduce a new form of concrete syntax, `fun f(x:`τ_1`):`τ_2 `=>` e, as "syntactic sugar". During parsing it is expanded into $\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$. This means that a `fun`-expression does not have first-class status. For example, we do not give any typing or evaluation rules since we type-check and evaluate the result of the syntactic expansion, not the original form.

At this point we can prove type preservation and value soundness for our language in the following form:

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$.
2. (Value Soundness) If $e \Downarrow v$ then v value.

There are some other properties of interest, such as $v \Downarrow v$ for any value v. Other natural properties do not hold. For example:

1. (Failure of Termination) There is an e such that $\cdot \vdash e : \tau$, but there is no v such that $e \Downarrow v$.
2. (Failure of Reverse Preservation) There are values v and expressions e such that $\cdot \vdash v : \tau$ and $e \Downarrow v$ but not $\cdot \vdash e : \tau$.

It is instructive to find such counterexamples and consider the reasons why neither termination nor reverse preservation can be expected for a practical programming language.

We will not prove either the positive or negative properties of our language in the form given above. The reason is that preservation, while certainly expected to hold, is somewhat weak as a language property: It only talks about expressions e that already are known to have a value. For example, if we omit the rule for function application (which is at the very heart of our language), then preservation would still hold! Moreover, any non-terminating computation is not addressed in this theorem at all.

This means we should look for stronger properties to characterize not only the relationship of an expression to its final value, but the process of computation itself. This requires a different form of operational semantics in which the steps of a computation are made explicit. We write $e \mapsto e'$ for the judgment that e steps to e', yet to be defined. It is related to the evaluation judgment in that

$$e \mapsto e_1 \mapsto \cdots \mapsto e_n \mapsto v \text{ for some } e_1, \ldots, e_n \text{ iff } e \Downarrow v$$

Before we give the rules, we state the properties we expect the language to satisfy in the end. This is a useful strategy which can prevent us from going astray and discovering potential problems with our judgments and rules early. The main properties are:

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$
2. (Progress) If $\cdot \vdash e : \tau$ then either
   (i) $e \mapsto e'$ for some e', or
   (ii) e value
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ and $e \mapsto e''$ then $e = e''$.

Usually, preservation and progress together are called *type safety*. Not all these properties are of equal importance, and we may have perfectly well-designed languages in which some of these properties fail. However, we want to clearly classify languages based on these properties and understand if they hold, or fail to hold.

Preservation. This is the most fundamental property, and it would be difficult to see how one could accept a type system in which this would fail.

Failure of this property amounts to a missing connection between the type system and the operational semantics, and it is unclear how we would even interpret the statement that $e : \tau$. If preservation holds, we can usually interpret a typing judgment as a partial correctness assertion about the expression:

> *If expression e has type τ and e evaluates to a value v, then v also has type τ.*

Progress. This property tells us that evaluation of an expression does not get stuck in any unexpected way: either we have a value (and are done), or there is a way to proceed. If a language is to satisfy progress it should not have any expressions whose operational meaning is undefined. For example, if we added division to MinML we could simply not specify any transition rule that would apply for the expression $\texttt{divide}(\texttt{num}(k), \texttt{num}(0))$. Not specifying the results of such a computation, however, is a bad idea because presumably an implementation will do *something*, but we can no longer know what. This means the behavior is implementation-dependendent and code will be unportable. To describe the behavior of such partial expressions we usually resort to introducing error states or exceptions into the language.

There are other situations where progress may be violated. For example, we may define a non-deterministic language that includes failure (non-deterministic choice between zero alternatives) as an explicit outcome.

Determinism. There are many languages, specifically those with concurrency or explicit non-deterministic choice, for which determinism fails, and for which it makes no sense to require it. On the other hand, we should always be aware whether our language is indeed deterministic or not. There are also situations where the language semantics explicitly violates determinism in order to give the language implementor the freedom to choose convenient strategies. For example, the *Revised*^{5} *Definition of Scheme*[1] states that the arguments to a function may be evaluated in any order. In fact, the order of evaluation for every single procedure call may be chosen differently!

While every implementation conforming to such a specification is presumably deterministic (and the language satisfies both preservation and progress), code which accidentally or consciously relies on the order of

[1] `http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html`

evaluation of a particular compiler will be non-portable between Scheme implementations. Moreover, the language provides absolutely no help in discovering such inadvisable implementation-dependence. While one is easily willing to accept this for concurrent languages, where different interleavings of computation steps are an unavoidable fact of life, it is unfortunate for a language which could quite easily be deterministic, and is intended to be used deterministically.

Small-step semantics. An operational semantics that specifies computation step by step is usually called a small-step semantics. We also call it *structural operational semantics*. We retain the value judgment defined in the last lecture and add the new judgment $e \mapsto e'$, as indicated above. When presenting the operational semantics, we proceed type by type.

Integers This is straightforward. We evaluate the arguments to a primitive operation from left to right, and apply the operation once all arguments have been evaluated.

$$\frac{e_1 \mapsto e_1'}{\texttt{equals}(e_1, e_2) \mapsto \texttt{equals}(e_1', e_2)} \qquad \frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{equals}(v_1, e_2) \mapsto \texttt{equals}(v_1, e_2')}$$

$$\frac{(k_1 = k_2)}{\texttt{equals}(\texttt{num}(k_1), \texttt{num}(k_2)) \mapsto \texttt{true}}$$

$$\frac{(k_1 \neq k_2)}{\texttt{equals}(\texttt{num}(k_1), \texttt{num}(k_2)) \mapsto \texttt{false}}$$

We refer to the first two as *search rules*, since they traverse the expression to "search" for the subterm where the actual computation step takes place. The latter two are *reduction rules*.

Booleans For if-then-else we have only one search rule for the condition, since we never evaluate in the branches before we know which one to take.

$$\frac{e \mapsto e'}{\texttt{if}(e, e_1, e_2) \mapsto \texttt{if}(e', e_1, e_2)}$$

$$\frac{}{\texttt{if}(\texttt{true}, e_1, e_2) \mapsto e_1} \qquad \frac{}{\texttt{if}(\texttt{false}, e_1, e_2) \mapsto e_2}$$

Definitions We proceed as in the expression language with the substitution semantics. There are no new values, and only one search rule.

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)}$$

$$\frac{v_1 \text{ value}}{\texttt{let}(v_1, x.e_2) \mapsto \{v_1/x\}e_2}$$

Functions Applications are evaluated from left-to-right, until both the function and its argument are values. This means the language is a *call-by-value* language with a *left-to-right* evaluation order.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)} \qquad \frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}$$

$$\frac{v_2 \text{ value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e), v_2) \mapsto \{v_2/x\}e}$$

Recursion A recursive expression is evaluated simply by unfolding it.

$$\frac{}{\texttt{rec}(\tau, x.e) \mapsto \{\texttt{rec}(\tau, x.e)/x\}e}$$

A recursive expression is never a value, but in a typical use of the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$$

we can make only one step before reaching a value, because unfolding the `rec` exposes an `fn`-abstraction which is always a value. In [Ch. 9], the recursive expression $\texttt{fun}(\tau_1, \tau_2, f.x.e)$ which corresponds to the above is directly a value. This is appropriate in the case of MinML, but would lead to difficulties in a more general setting later in the course where we study recursively defined lists, trees, and other data structures.

Preservation. For the proof of preservation we need two properties about the substitution operation as it occurs in the cases of `let`-expressions and function application. We state them here in a slightly more general form than we need, but a slightly less general form than what is possible.

Theorem 1 (Properties of Typing)

(i) (Weakening) If $\Gamma_1, \Gamma_2 \vdash e' : \tau'$ *the* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$.

(ii) (Substitution)
If $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$ *and* $\cdot \vdash e : \tau$ *then* $\Gamma_1, \Gamma_2 \vdash \{e/x\}e' : \tau'$.

Proof: Property *(i)* follows directly by rule induction on the given derivation: we can insert the additional hypothesis in every hypothetical judgment occuring in the derivation without invalidating any rule applications.

Property *(ii)* allso follows by a rule induction on the given derivation of $\Gamma_1, x{:}\tau, \Gamma_2 \vdash e' : \tau'$. Since typing and substitution are both compositional over the structure of the term, the only interesting cases is where e' is the variable x.

Case: (Rule *VarTyp*) with $e' = x$**.** Then $\tau' = \tau$ and $\{v/x\}e' = \{v/x\}x = v$. So we have to show $\Gamma_1, \Gamma_2 \vdash v : \tau$. But our assumption is $\cdot \vdash v : \tau$ so we can conclude this by weakening (Property *(i)*). ■

Both the weakening and substitution properties arise directly from the nature of reasoning from assumption. They are special cases of very general properties of hypothetical judgments.

Weakening is a valid principle, because when we reason from assumption nothing compels us to actually use any given assumption. Therefore we can always add more assumptions without invalidating our conclusion.

Substitution is a valid principle, because we can always replace the use of an assumption by its derivation.

The proof below requires the use of the proof principle of *inversion*. Say in the course of a proof you have established that a certain judgment J has a derivation. If you can see, purely syntactically, that there is only one possible inference rule that could have been used to conclude J, then we know the premises of the rule must also be hold. It is called inversion because in a strange way we go from a derivation of the conclusion to a derivation of the premises. The proof of preservation below uses inversion essentially in each case, applying it to the given typing derivation for e. Since the typing judgment is syntax-directed, and there is exactly one rule for each kind of expression, it is usually straightforward to apply inversion.

A word of caution: many mistakes arise in proofs because inversion is used incorrectly. Remember: you can only apply it if you already know, either from and assumption or the induction hypothesis, that a certain jnudgment must have a derivation.

Theorem 2 (Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the derivation of $e \mapsto e'$. In each case we apply inversion to the given typing derivation and then apply either the induction hypothesis or directly construct a typing derivation for e'.

Critical in this proof is the syntax-directed nature of the typing rules: for each construct in the language there is exactly one typing rule. Preservation is significantly harder for languages that do not have this property, and there are many advanced type systems that are *not* a priori syntax-directed.

We only show the cases for booleans and functions, leaving integers and `let`-expressions to the reader.

Case

$$\frac{e_1 \mapsto e_1'}{\texttt{if}(e_1, e_2, e_3) \mapsto \texttt{if}(e_1', e_2, e_3)}$$

This case is typical for search rules, which compute on some subexpression.

$e_1 \mapsto e_1'$	Subderivation
$\cdot \vdash \texttt{if}(e_1, e_2, e_3) : \tau$	Assumption
$\cdot \vdash e_1 : \texttt{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_1' : \texttt{bool}$	By i.h.
$\cdot \vdash \texttt{if}(e_1', e_2, e_3) : \tau$	By rule

Case

$$\frac{}{\texttt{if}(\texttt{true}, e_2, e_3) \mapsto e_2}$$

$\cdot \vdash \texttt{if}(\texttt{true}, e_2, e_3) : \tau$	Assumption
$\cdot \vdash \texttt{true} : \texttt{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e_2 : \tau$	In line above

Case

$$\frac{}{\texttt{if}(\texttt{false}, e_2, e_3) \mapsto e_3}$$

Symmetric to the previous case.

Case

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}$$

$e_1 \mapsto e_1'$	Subderivation
$\cdot \vdash \texttt{apply}(e_1, e_2) : \tau$	Assumption
$\cdot \vdash e_1 : \texttt{arrow}(\tau', \tau)$ and $\cdot \vdash e_2 : \tau'$ for some τ'	By inversion
$\cdot \vdash e_1' : \texttt{arrow}(\tau', \tau)$	By i.h.
$\cdot \vdash \texttt{apply}(e_1', e_2) : \tau$	By rule

Case

$$\frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}$$

Analogous to the previous case.

Case

$$\frac{v_2 \text{ value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}$$

$\cdot \vdash \texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) : \tau$	Assumption
$\cdot \vdash \texttt{fn}(\tau_2, x.e_1) : \texttt{arrow}(\tau', \tau)$ and $\cdot \vdash v_2 : \tau'$ for some τ'	By inversion
$\cdot, x{:}\tau' \vdash e_1 : \tau$ and $\tau_2 = \tau$	By inversion
$\cdot \vdash \{v_2/x\}e_1 : \tau$	By substitution property

Case

$$\frac{}{\texttt{rec}(\tau', x.e') \mapsto \{\texttt{rec}(\tau', x.e')/x\}e'}$$

$\cdot \vdash \texttt{rec}(\tau', x.e') : \tau$	Assumption
$\cdot, x{:}\tau \vdash e' : \tau$ and $\tau' = \tau$	By inversion
$\cdot \vdash \{\texttt{rec}(\tau, x.e')/x\}e'$	By substitution property

■

In summary, in MinML preservation comes down to two observations: (1) for the search rules, we just use the induction hypothesis, and (2) for reduction rules, the interesting cases rely on the substitution property. The latter states that substuting a (closed) expression of type τ for a variable of

type τ in an expression of type τ' preserves the type of that expression as τ'.

In the next lecture we show the proof of the progress theorem and also extend our language with more type constructors that will be necessary to represent more complex data types.

Lectures Notes on Progress

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 7
September 21, 2004

In this lecture we prove the progress property for MinML, discuss type safety, and consider which other language features may be desirable or undesirable in a language definition. We also consider how we have to change the operational semantics and the statement of the progress theorem when run-time errors are permitted in the language, such as division by zero. As a reminder, type safety consists of preservation (proved in the last lecture) and progress in the following form.

1. (Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$

2. (Progress) If $\cdot \vdash e : \tau$ then either

   (i) $e \mapsto e'$ for some e', or

   (ii) e value

3. (Determinism) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ and $e \mapsto e''$ then $e' = e''$.

Determinism is of particular interest for sequential languages, where we generally expect it to hold.

Not all these properties are of equal importance, and we may have perfectly well-designed languages in which some of these properties fail. However, we want to clearly classify languages based on these properties and understand if they hold, or fail to hold. Please consult the notes of the previous lecture for a further discussion of some of these issues.

Progress. We now turn our attention to the progress theorem. This asserts that the computation of closed well-typed expressions will never get stuck, although it is quite possible that it does not terminate. For example,

$$\texttt{rec}(\texttt{int}, x.x)$$

reduces in one step to itself.

The critical observation behind the proof of the progress theorem is that a value of function type will indeed be a function, a value of boolean type will indeed by either `true` or `false`, etc. If that were not the case, then we might reach an expression such as

$$\texttt{apply}(\texttt{num}(0), \texttt{num}(1))$$

which is a stuck expression because $\texttt{num}(0)$ and $\texttt{num}(1)$ are values, so neither any of the search rules nor the reduction rule for application can be applied. We state these critical properties as an inversion lemmas, because they are not immediately syntactically obvious.

Lemma 1 (Value Inversion)

(i) If $\cdot \vdash v : \texttt{int}$ *and* v value *then* $v = \texttt{num}(n)$ *for some integer* n.

(ii) If $\cdot \vdash v : \texttt{bool}$ *and* v value *then* $v = \texttt{true}$ *or* $v = \texttt{false}$.

(iii) If $\cdot \vdash v : \texttt{arrow}(\tau_1, \tau_2)$ *and* v value *then* $v = \texttt{fn}(\tau_1, x.e)$ *for some* $x.e$.

Proof: We distinguish cases on v value and then apply inversion to the given typing judgment. We show only the proof of property *(ii)*.

Case: $v = \texttt{num}(n)$**.** Then we would have $\cdot \vdash \texttt{num}(n) : \texttt{bool}$, which is impossible by inspection of the typing rules.

Case: $v = \texttt{true}$**.** Then we are done, since, indeed $v = \texttt{true}$ or $v = \texttt{false}$.

Case: $v = \texttt{false}$**.** Symmetric to the previous case.

Case: $v = \texttt{fn}(\tau, x.e)$**.** As in the first case, this is impossible by inspection of the typing rules. ■

The preceding value inversion lemmas is also called the *canonical forms theorem* [Ch. 10.2]. Now we can prove the progress theorem.

Theorem 2 (Progress)
If $\cdot \vdash e : \tau$ *then*

(i) either $e \mapsto e'$ *for some* e'*,*

(ii) or e value*.*

Proof: By rule induction on the given typing derivation. Again, we show only the cases for booleans and functions.

Case

$$\frac{x{:}\tau \in \cdot}{\cdot \vdash x : \tau}\ \textit{VarTyp}$$

This case is impossible since the context is empty.

Case

$$\frac{}{\cdot \vdash \texttt{true} : \texttt{bool}}\ \textit{TrueTyp}$$

Then `true` value.

Case

$$\frac{}{\cdot \vdash \texttt{false} : \texttt{bool}}\ \textit{FalseTyp}$$

Then `false` value.

Case

$$\frac{\cdot \vdash e_1 : \texttt{bool} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{\cdot \vdash \texttt{if}(e_1, e_2, e_3) : \tau}\ \textit{IfTyp}$$

In this case it is clear that $\texttt{if}(e_1, e_2, e_3)$ cannot be a value, so we have to show that $\texttt{if}(e_1, e_2, e_3) \mapsto e'$ for some e'.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 value	By i.h.
$e_1 \mapsto e_1'$	First subcase
$\texttt{if}(e_1, e_2, e_3) \mapsto \texttt{if}(e_1', e_2, e_3)$	By rule
e_1 value	Second subcase
$e_1 = \texttt{true}$ or $e_1 = \texttt{false}$	By value inversion
$e_1 = \texttt{true}$	First subsubcase
$\texttt{if}(\texttt{true}, e_2, e_3) \mapsto e_2$	By rule
$e_1 = \texttt{false}$	Second subsubcase
$\texttt{if}(\texttt{false}, e_2, e_3) \mapsto e_3$	By rule

Case

$$\frac{\cdot, x{:}\tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \mathtt{fn}(\tau_1, x.e_2) : \mathtt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

Then $\mathtt{fn}(\tau_1, x.e_2)$ value.

Case

$$\frac{\cdot \vdash e_1 : \mathtt{arrow}(\tau_2, \tau) \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 value	By i.h.
$e_1 \mapsto e_1'$	First subcase
$\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1', e_2)$	By rule
e_1 value	Second subcase
Either $e_2 \mapsto e_2'$ for some e_2' or e_2 value	By i.h.
$e_2 \mapsto e_2'$	First subsubcase
$\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1, e_2')$	By rule (since e_1 value)
e_2 value	Second subsubcase
$e_1 = \mathtt{fn}(\tau_2, x.e_1')$	By value inversion
$\mathtt{apply}(e_1, e_2) \mapsto \{e_2/x\}e_1'$	By rule (since e_2 value)

Case

$$\frac{\cdot, x{:}\tau \vdash e' : \tau}{\cdot \vdash \mathtt{rec}(\tau, x.e') : \tau}\ \textit{RecTyp}$$

$\mathtt{rec}(\tau, x.e') \mapsto \{\mathtt{rec}(\tau, x.e')/x\}e'$	By rule

■

Determinism. We will leave the proof of determinism to the reader—it is not difficult given all the examples and techniques we have seen so far.

Call-by-Value vs. Call-by-Name. The MinML language as described so far is a *call-by-value* language because the argument of a function call is

evaluated before passed to the function. This is captured the following rules.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}\ cbv.1$$

$$\frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}\ cbv.2$$

$$\frac{v_2 \text{ value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ cbv.f$$

We can create a call-by-name variant by *not* permitting the evaluation of the argument (rule $cbv.2$ disappears), but just passing it into the function (replace $cbv.r$ by $cbn.r$). The first rule just carries over.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)}\ cbn.1$$

$$\frac{}{\texttt{apply}(\texttt{fn}(\tau_1, x.e_1), e_2) \mapsto \{e_2/x\}e}\ cbn.f$$

Evaluation Order. Our specification of MinML requires the we first evaluate e_1 and then e_2 in application $\texttt{apply}(e_1, e_2)$. We can also reduce from right to left by switching the two search rules. The last one remains the same.

$$\frac{e_2 \mapsto e_2'}{\texttt{apply}(e_1, e_2') \mapsto \texttt{apply}(e_1, e_2')}\ cbvr.1$$

$$\frac{e_1 \mapsto e_1' \quad v_2 \text{ value}}{\texttt{apply}(e_1, v_2) \mapsto \texttt{apply}(e_1', v_2)}\ cbvr.2$$

$$\frac{v_2 \text{ value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e_1), v_2) \mapsto \{v_2/x\}e_1}\ cbvr.f$$

The O'Caml dialect of ML indeed evaluates from right-to-left, while Standard ML evaluates from left-to-right. There does not seem to be an intrinsic reason to prefer one over the other, except perhaps that evaluating a term in the order it is written appears slightly more natural.

Accounting for Errors [1] It is not always possible to avoid run-time errors, due to limitations in type systems. To illustrate how they can be accounted

[1] This section adapted from notes by Daniel Spoonhower, Fall 2003.

for we will add another primitive operator over integers, division. Unlike addition, subtraction, and multiplication, the division of integers is a *partial* function. That is, it does not yield a result for all possible inputs. In particular, consider the expression `div(num(2), num(0))`. We would like to include division in our type-safe language, but so far we have no way of accounting for what "happens" when we evaluate a division by zero.

(One possibility is to add an additional value of type `int` that is the result of such an expression. This value is sometimes called "NaN" or "not-a-number" when it appears in specifications of floating-point arithmetic. If we were to do so, however, we would have other problems to consider; for example, what is the result of `num(1) = NaN`?)

We will add a new expression to our language, shown below, to capture the state when an expression is "undefined". (This expression is also sometimes known `wrong` or as the "stuck state.")

$$e ::= \ldots \mid \texttt{error}$$

(Is `error` a value? Why or why not? It may become more clear when we introduce a typing rule for `error` below.)

With `error` in hand, we can give an evaluation rule that applies to the expression above.

$$\frac{}{\texttt{div(num(}k\texttt{), num(0))} \mapsto \texttt{error}}\ DivZero$$

We haven't quite finished with evaluation yet, however: consider the following expression:

$$\texttt{if(div(num(2), num(0)),...)} \mapsto \texttt{if(error,...)} \mapsto ?$$

Even though we've made progress with division, we still are stuck at the `if`. We will need to add new rules to *propagate* errors through all of our existing constructs. Analogously to our search evaluation rules, we add:

$$\frac{}{\texttt{apply(error,}e_2\texttt{)} \mapsto \texttt{error}} \qquad \frac{v_1 \text{ value}}{\texttt{apply(}v_1\texttt{,error)} \mapsto \texttt{error}}$$

$$\frac{}{\texttt{if(error,}e_1\texttt{,}e_2\texttt{)} \mapsto \texttt{error}} \qquad \frac{}{\texttt{let(error,}x.e\texttt{)} \mapsto \texttt{error}}$$

$$\frac{v_1 \text{ value} \quad \ldots \quad v_{j-1} \text{ value}}{o\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_{j-1}\texttt{,error,}e_{j+1}\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mapsto \texttt{error}}$$

Here, o stands for a primitive operations with n arguments.

Typing For Errors Before we can go ahead and extend our safety proof, we must give a type to our new expression. Since no actual computation is performed once we have encountered an `error`, we can assign *any* type to an expression that has failed (i.e., there is no way to distinguish one error from another).

$$\frac{}{\Gamma \vdash \texttt{error} : \tau}\ ErrorTyp$$

Preservation

If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$. We have previously shown this proof by induction over the derivation of $e \mapsto e'$, so we have six new cases to consider. We show only two.

Rule *DivZero* $e' = \texttt{error}$

There are no assumptions to this rule, so we have no subderivations to consider. However, we only need to show that $\cdot \vdash e' : \tau$. Since $e' = \texttt{error}$, this is easy enough.

$\cdot \vdash \texttt{error} : \tau$ By rule

Rule *IfError* $e' = \texttt{error}$

Again we have no assumptions and so, again, no subderivations. In fact, this case looks just like the last case!

$\cdot \vdash \texttt{error} : \tau$ By rule

All of our new cases for preservation look exactly like this since each evaluates (in one step) to the `error` expression. With these new cases, our extended proof of preservation is complete.

Progress

Here we must extend the theorem: if $\cdot \vdash e : \tau$ then either

i. e value or

ii. $e \mapsto e'$ for some e' or

iii. e is `error`

This proof was given by rule induction over the derivation of $\cdot \vdash e : \tau$, and we have one new typing rule to consider, so we have one additional case.

Rule *ErrorTyp* $e = \texttt{error}$

e is `error` By assumption

Easy enough! Have we finished? No, because we have extended the induction hypothesis, we have an additional subcase to consider each time we applied it.

Consider the case for *IfTyp*:

$$\frac{\begin{array}{ccc}\vdots & \vdots & \vdots \\ \cdot \vdash e : \texttt{bool} & \cdot \vdash e_1 : \tau & \cdot \vdash e_2 : \tau\end{array}}{\cdot \vdash \texttt{if}(e, e_1, e_2) : \tau}$$

Previously, we applied the induction hypothesis to the first subderivation to conclude:

$$\text{Either } e \text{ value or } e \mapsto e'$$

Now must must consider each of:

$$\text{Either } e \text{ value or } e \mapsto e' \text{ or } e \text{ is } \texttt{error}$$

The first two subcases are identical to those in our old proof, but we must finish the third.

e is `error` By case (iii) of i.h.
$\texttt{if}(\texttt{error}, e_1, e_2) \mapsto \texttt{error}$ By rule

We have shown that there is a step to be made and so progress is maintained.

In each of the applications of the induction hypothesis, we will have a new subcase, and (if we've set things up correctly) we should have a new rule to apply. If we find a subcase and no rule to apply, it probably means that we've forgotten a rule; conversely, if a new rule doesn't apply anywhere, it was probably unnecessary.

(Is it clear now why we don't want `error` to be a value? Think about value inversion with respect to `error`.)

Lecture Notes on Aggregate Data Structures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 8
September 23, 2004

In this lecture we discuss various language extensions which make MinML a more realistic language without changing its basic character. In the second part of the lecture we also consider an environment-based semantics in which we avoid explicit application of substitution to give the semantics a more realistic character and discuss some common mistakes in language definition.

Products. Introducing products just means adding pairs and a unit element to the language [Ch. 19.1]. We could also directly add n-ary products, but we will instead discuss records later when we talk about object-oriented programming. MinML is a call-by-value language. For consistency with the basic choice, the pair constructor also evaluates its arguments—otherwise we would be dealing with *lazy pairs*.[1] In addition to the `pair` constructor, we can extract the first and second component of a pair.[2]

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1, e_2) : \mathtt{cross}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \mathtt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \mathtt{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \mathtt{cross}(\tau_1, \tau_2)}{\Gamma \vdash \mathtt{snd}(e) : \tau_2}$$

We often adopt a more mathematical notation according to the table at the end of these notes. However, it is important to remember that the

[1] See Assignment 3

[2] An alternative treatment is given in [Ch. 19.1], where the destructor provides access to both components of a pair simultaneously.

mathematical shorthand is just that: it is just a different way to shorten higher-order abstract syntax or make it easier to read.

A pair is a value if both components are values. If not, we can use the search rules to reduce, using a left-to-right order. Finally, the reduction rules extract the corresponding component of a pair.

$$\frac{e_1 \text{ value} \quad e_2 \text{ value}}{\texttt{pair}(e_1, e_2) \text{ value}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{pair}(e_1, e_2) \mapsto \texttt{pair}(e_1', e_2)} \qquad \frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{pair}(v_1, e_2) \mapsto \texttt{pair}(v_1, e_2')}$$

$$\frac{e \mapsto e'}{\texttt{fst}(e) \mapsto \texttt{fst}(e')} \qquad \frac{e \mapsto e'}{\texttt{snd}(e) \mapsto \texttt{snd}(e')}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{fst}(\texttt{pair}(v_1, v_2)) \mapsto v_1} \qquad \frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{snd}(\texttt{pair}(v_1, v_2)) \mapsto v_2}$$

Since it is at the core of the progress property, we make the value inversion property explicit.

> If $\cdot \vdash v : \texttt{cross}(\tau_1, \tau_2)$ and v value then $v = \texttt{pair}(v_1, v_2)$ for some v_1 value and v_2 value.

Unit Type. For the unit type we only have a constructor but no destructor, since there are no components to extract.[3]

$$\frac{}{\Gamma \vdash \texttt{unitel} : \texttt{unit}}$$

The unit types does not yield any new search or reduction rules, only a new value. At first it may not seem very useful, but we will see an application when we add references to the language.

$$\frac{}{\texttt{unitel} \text{ value}}$$

The value inversion property is also simple.

> If $\cdot \vdash v : \texttt{unit}$ then $v = \texttt{unitel}$.

[3] A so-called `check` construct is possible but not necessary; see [Ch. 19.1].

Sums. Unions, as one might know them from the C programming language, are inherently not type safe. They can be abused in order to access the underlying representations of data structures and intentionally violate any kind of abstraction that might be provided by the language. Consider, for example, the following snippet from C.

```
union {
   float f;
   int   i;
} unsafe;

unsafe.f = 5.67e-5;
printf("%d", unsafe.i);
```

Here we set the member of the union as a floating point number and then print the underlying bit pattern as if it represented an integer. Of course, much more egregious examples can be imagined here.

In a type-safe language we replace unions by disjoint sums. In the implementation, the members of a disjoint sum type are tagged with their origin so we can safely distinguish the cases. In order for every expression to have a unique type, we also need to index the corresponding injection operator with their target type. We avoid this complication here, postponing the issue of how to perform type-checking to a future lecture.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{inl}(e_1) : \mathtt{sum}(\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{inr}(e_2) : \mathtt{sum}(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \mathtt{sum}(\tau_1, \tau_2) \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) : \sigma}$$

Note that we require both branches of a `case`-expression to have the same type σ, just as for a conditional, because we cannot be sure at type-checking

time which branch will be taken at run time.

$$\frac{e_1 \text{ value}}{\texttt{inl}(e_1) \text{ value}} \qquad \frac{e_2 \text{ value}}{\texttt{inr}(e_2) \text{ value}}$$

$$\frac{e \mapsto e'}{\texttt{case}(e, x_1.e_1, x_2.e_2) \mapsto \texttt{case}(e', x_1.e_1, x_2.e_2)}$$

$$\frac{v_1 \text{ value}}{\texttt{case}(\texttt{inl}(v_1), x_1.e_1, x_2.e_2) \mapsto \{v_1/x_1\}e_1}$$

$$\frac{v_2 \text{ value}}{\texttt{case}(\texttt{inr}(v_2), x_1.e_1, x_2.e_2) \mapsto \{v_2/x_2\}e_2}$$

We also state the value inversion property.

> If $\cdot \vdash v : \texttt{sum}(\tau_1, \tau_2)$ then either $v = \texttt{inl}(v_1)$ with v_1 value or $v = \texttt{inr}(v_2)$ with v_2 value.

Void type. The empty type `void` can be thought of as a zero-ary sum. It has no values, and can only be given to expressions that do not terminate. For example,

$$\frac{\Gamma, x{:}\texttt{void} \vdash x : \texttt{void}}{\Gamma \vdash \texttt{rec}(\texttt{void}, x.x) : \texttt{void}}$$

The value inversion property here just expresses that there are no values of void type.

> If $\cdot \vdash v : \texttt{void}$ then we have a contradiction.

In this lecture we did not explicitly revisit the cases in the proof of the preservation and progress theorem, but the cases follow exactly the previously established patterns.

Environment-Based Semantics. So far, most of our semantic specifications rely on *substitution* as a primitive operation. From the point of view of implementation, this is impractical, because a program would be copied many times. So we seek an alternative semantics in which substitutions are not carried out explicitly, but an association between variables and their values is maintained. Such a data structure is called an *environment*. Care has to be taken to ensure that the intended meaning of the program (as

given by the specification with substitution) is not changed. We have already discussed such a semantics in Lecture 4 for arithmetic expressions.

Because we are in a call-by-value language, environments η bind variables to values.

$$\text{Environments} \quad \eta \quad ::= \quad \cdot \mid \eta, x{=}v$$

The basic intuition regarding typing is that if $\Gamma \vdash e : \tau$, then e should be evaluated in an environment which supplies bindings of appropriate type for all the variables declared in Γ. We therefere formalize this as a judgment, writing $\eta : \Gamma$ if the bindings of variables to values in η match the context Γ. We make the general assumption that a variable x is bound only once in an environment, which corresponds to the assumption that a variable x is declared only once in a context. If necessary, we can rename bound variables in order to maintain this invariant.

$$\frac{}{\cdot : \cdot} \qquad \frac{\eta : \Gamma \quad \cdot \vdash v : \tau \quad v \ \mathsf{value}}{(\eta, x{=}v) : (\Gamma, x{:}\tau)}$$

Note that the values v bound in an environment are closed, that is, they contain no free variables. This means that expressions are evaluated in an environment, but the resulting values must be closed. This creates a difficulty when we come to the evaluation of function expressions. Relaxing this restriction, however, causes even more serious problems.[4]

We start with integers and let-bindings, the latter of which is an explicit motivation for the introduction of environments. Here we assume some primitive operators o (such as `plus` and `times`) and their mathematical counterparts f_o. For simplicity, we just write binary operators here.

$$\frac{x{\Downarrow}v \in \eta}{\eta \vdash x \Downarrow v}\ e.var \qquad \frac{}{\eta \vdash \mathsf{num}(k) \Downarrow \mathsf{num}(k)}\ e.num$$

$$\frac{\eta \vdash e_1 \Downarrow \mathsf{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathsf{num}(k_2) \quad (f_o(k_1, k_2) = k)}{\eta \vdash \mathsf{o}(e_1, e_2) \Downarrow \mathsf{num}(k)}\ e.o$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x{\Downarrow}v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2}\ e.let \quad (x \text{ not declared in } \eta)$$

Next we come to functions. Before we state these rules let's explicitly state the preservation property we expect to hold at the end. Note that there is no progress property, because it cannot be formulated very easily on the big-step semantics.

[4] This is known in the Lisp community as the *upward funarg problem*.

Preservation. If $\Gamma \vdash e : \tau$ and $\eta : \Gamma$ and $\eta \vdash e \Downarrow v$ then $\cdot \vdash v : \tau$.

Note in particular here the formal expression of the intuition above that the output v must be closed. The following rule

$$\frac{}{\eta \vdash \mathtt{fn}(\tau, x.e) \Downarrow \mathtt{fn}(\tau, x.e)}\ e.fn?$$

would be incorrect because $\mathtt{fn}(\tau, x.e)$ can refer to variables defined in η which would "leak" into the output value, violating the closedness condition of the preservation theorem. Instead we need to create a so-called *closure* which pairs up a function with its environment, representing a new form of value. We write

$$\langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle$$

for the closure of $\mathtt{fn}(\tau, x.e)$ over the environment η.

There are no evaluation rules for closures (they are values), and the typing rules have to "guess" a context that matches the environment. Note that we always type values in the empty environment.

$$\frac{}{\langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle\ \ \mathsf{value}} \qquad \frac{\eta : \Gamma \quad \Gamma \vdash \mathtt{fn}(\tau, x.e) : \tau'}{\cdot \vdash \langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle : \tau'}$$

Note that function expressions like $\mathtt{fn}(\tau, x.e)$ are no longer values–only function closures are values. We now modify the incorrect rule by building a closure instead and write down the right evaluation rule for function application.

$$\frac{}{\eta \vdash \mathtt{fn}(\tau, x.e) \Downarrow \langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle}\ e.fn$$

$$\frac{\eta \vdash e_1 \Downarrow \langle\!\langle \eta'; \mathtt{fn}(\tau_2, x.e_1') \rangle\!\rangle \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta', x{=}v_2 \vdash e_1' \Downarrow v}{\eta \vdash \mathtt{apply}(e_1, e_2) \Downarrow v}\ e.app$$

Note that every aspect of this rule is critical: evaluation of e_1 returns a closure instead of a function expression, and the body of the function is evaluated in environment found in the closure extended by the binding for x. By our general convention about variables, x may not already be declared in the environment η' so the new one is well-formed. This can always be achieved by the tacit renaming of the bound variable so it differens from the variables in η'.

One interesting problems that arises in this context is the treatment of recursion. There are a number of ways to avoid explicit substitutions, such

as creating recursive environments, or allowing bindings of variables to unevaluated expressions, to be evaluated when they are looked up. The solution taken in Standard ML is to syntactically restrict recursion to functional expressions and declare the resulting functions $\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$ to be values. This is the solution taken in the notes [Ch. 9].

Higher-Order Abstract Syntax	Concrete Syntax	Mathematical Syntax
`arrow`(τ_1, τ_2)	τ_1 `->` τ_2	$\tau_1 \rightarrow \tau_2$
`cross`(τ_1, τ_2)	τ_1`*`τ_2	$\tau_1 \times \tau_2$
`unit`	`unit`	1
`sum`(τ_1, τ_2)	τ_1`+`τ_2	$\tau_1 + \tau_2$
`void`	`void`	0
`pair`(e_1, e_2)	`(`e_1`,`e_2`)`	$\langle e_1, e_2 \rangle$
`fst`(e)	`#1` e	$\pi_1\, e$
`snd`(e)	`#2` e	$\pi_2\, e$
`unitel`	`()`	$\langle\,\rangle$
`inl`(e_1)	`inl`(e_1)	$\mathsf{inl}_{\tau_1+\tau_2}(e_1)$
`inr`(e_2)	`inr`(e_2)	$\mathsf{inr}_{\tau_1+\tau_2}(e_2)$
`case`$(e, x_1.e_1, x_2.e_2)$	`case` e `of` `inl`(x_1) `=>` e_1 `|` `inr`(x_2) `=>` e_2 `esac`	$\mathsf{case}(e, x_1.e_1, x_2.e_2)$
`abort`(e)	`abort`(e)	$\mathsf{abort}_\tau(e)$

Supplementary Notes on An Abstract Machine

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 9
Sep 28, 2004

In this lecture we introduce a somewhat lower-level semantics for MinML in the form of an *abstract machine* [Ch. 11]. In this machine we make the control flow explicit, rather than encoding it in the search rules as in the first operational semantics. Besides getting closer to an actual implementation, it will allow us to easily define constructs to capture the current continuation [Ch. 12].

Abstract machines have recently gained in popularity through the ascendency of the Java programming language. The standard model is that we compile Java source to Java bytecode, which may be transmitted over networks (for example, as an "applet"), and then interpreted via the Java abstract machine. The use of an abstract machine here plays two important roles: (1) the byte code is portable to any architecture with an interpreter, and (2) the received code can be easily checked for illegal operations. This is type-checking of the abstract machine code goes hand in hand with some residual checking that has to go on while the code is interpreted. Note that traditional type-checking as we have discussed it so far needs to be augmented significantly, for example, to prevent the normally type-safe operation of reformatting the hard disk.

The kind of abstract machine we present here is a variant of the C-machine [Ch. 11.1] with two kinds of states: those that attempt to evaluate an expression, and those that return a value that has been computed. Its main component, however, is the same: a run-time stack that records what remains to be done after the current subexpression has been fully evaluated. The stack consists of frames which represent the action to be taken by the abstract machine once the current expression has been evaluated. We treat here the fragment with pairs, functions, and booleans (see [Ch. 11.1]

for a treatment of primitive operators).

We begin by defining the syntax in the form of (abstract syntax) grammar. As we have seen before, this can also be written in the form of judgments. When we use v we imply that v must be a value.

States	s	$::=$	$k > e$	evaluate e under k
		$\mid$	$k < v$	return v to k
Stacks	k	$::=$	$\bullet$	empty stack
		$\mid$	$k \triangleright f$	stack k with top f
Frames	f	$::=$	$\mathtt{o}(\square, e_2) \mid \mathtt{o}(v_1, \square)$	primops
		$\mid$	$\mathtt{pair}(\square, e_2) \mid \mathtt{pair}(v_1, \square)$	pairs
		$\mid$	$\mathtt{fst}(\square) \mid \mathtt{snd}(\square)$	projections
		$\mid$	$\mathtt{apply}(\square, e_2) \mid \mathtt{apply}(v_1, \square)$	applications
		$\mid$	$\mathtt{if}(\square, e_1, e_2)$	conditional

A hole $\square$ in the top stack frame is intended to hold the value returned by evaluation of the current expression. It corresponds to the place in an expression where evaluation can take place and thus implements the search rules of the structured operational semantics.

The main judgment defining the abstract machine is

$$s \mapsto_{\mathsf{c}} s'$$

expressing that state s makes a transition to state s' in one step. The initial state of the machine has the form $\bullet > e$, a final state has the form $\bullet < v$. In general, we define our machine so that if

$$e = e_1 \mapsto \cdots \mapsto e_n = v$$

according to our operational semantics then for any stack k which should have

$$k > e \mapsto_{\mathsf{c}} \cdots \mapsto_{\mathsf{c}} k < v$$

As we will see, the operational semantics and the abstract machine do not take the same number of steps. This is because the operational semantics does not step at all for values, while the abstract machine will take some steps to go from $k > v$ to $k < v$.

Before we give the transitions of the C-machine, it is useful to think about typing and which properties besides the operational ones above we want to hold. First, we need to type states. A state $k > e$ should require that (1) e is closed (since we are evaluating it), (2) that e is well-typed, say, of

type τ, and (3) that k is a stack that expects a value of type τ to be returned to it. We also keep track of the type of the final result returned when both e and k are finished. Finally, a frame accepts a value (to be placed in its hole) and eventually passes value to the rest of the stack. These considerations yields the following typing judgments

$s : \sigma$	state s returns a final answer of type σ
$k : \tau \Rightarrow \sigma$	stack k expects a value of type τ and returns a final answer of type σ
$f : \tau \Rightarrow \sigma$	frame r expects a value of type τ and computes a value of type σ

We use the notation $\tau \Rightarrow \sigma$ as a suggestive notation, but you should keep in mind that frames f are not formally functions in our semantics. However, frames and stack can be formally related to functions, but we will not make this relationship explicit here.

With these definition, we can write out the rules. We have added some parentheses to make the reading of the judgments less ambiguous.

$$\frac{k : \tau \Rightarrow \sigma \quad \cdot \vdash e : \tau}{(k > e) : \sigma} \qquad \frac{k : \tau \Rightarrow \sigma \quad \cdot \vdash v : \tau \quad v \text{ value}}{(k < v) : \sigma}$$

$$\frac{}{\bullet : \tau \Rightarrow \tau} \qquad \frac{k : \tau' \Rightarrow \sigma \quad f : \tau \Rightarrow \tau'}{(k \triangleright f) : \tau \Rightarrow \sigma}$$

We show the typing rules for the individual frames as they are introduced in the operational semantics below.

We now give the transitions, organized by the type structure of the language.

Integers.

$$\begin{array}{lll} k > \mathsf{num}(n) & \mapsto_{\mathsf{c}} & k < \mathsf{num}(n) \\ \\ k > \mathsf{o}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \mathsf{o}(\Box, e_2) > e_1 \\ k \triangleright \mathsf{o}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \mathsf{o}(v_1, \Box) > e_2 \\ k \triangleright \mathsf{o}(\mathsf{num}(n_1), \Box) < \mathsf{num}(n_2) & \mapsto_{\mathsf{c}} & k < \mathsf{num}(n) \\ & & \quad (n = f_o(n_1, n_2)) \end{array}$$

$$\frac{\cdot \vdash e_2 : \mathtt{int}}{\mathsf{o}(\Box, e_2) : \mathtt{int} \Rightarrow \mathtt{int}} \qquad \frac{\cdot \vdash v_1 : \mathtt{int} \quad v_1 \text{ value}}{\mathsf{o}(v_1, \Box) : \mathtt{int} \Rightarrow \mathtt{int}}$$

Products.

$$
\begin{array}{lcl}
k > \texttt{pair}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{pair}(\Box, e_2) > e_1 \\
k \triangleright \texttt{pair}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \texttt{pair}(v_1, \Box) > e_2 \\
k \triangleright \texttt{pair}(v_1, \Box) < v_2 & \mapsto_{\mathsf{c}} & k < \texttt{pair}(v_1, v_2)
\end{array}
$$

$$
\begin{array}{lcl}
k > \texttt{fst}(e) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{fst}(\Box) > e \\
k \triangleright \texttt{fst}(\Box) < \texttt{pair}(v_1, v_2) & \mapsto_{\mathsf{c}} & k < v_1
\end{array}
$$

$$
\begin{array}{lcl}
k > \texttt{snd}(e) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{snd}(\Box) > e \\
k \triangleright \texttt{snd}(\Box) < \texttt{pair}(v_1, v_2) & \mapsto_{\mathsf{c}} & k < v_2
\end{array}
$$

$$
\frac{\cdot \vdash e_2 : \tau_2}{\texttt{pair}(\Box, e_2) : \tau_1 \Rightarrow \tau_1 \times \tau_2}
\qquad
\frac{\cdot \vdash v_1 : \tau_1 \quad v_1 \text{ value}}{\texttt{pair}(v_1, \Box) : \tau_2 \Rightarrow \tau_1 \times \tau_2}
$$

$$
\frac{}{\texttt{fst}(\Box) : \tau_1 \times \tau_2 \Rightarrow \tau_1}
\qquad
\frac{}{\texttt{snd}(\Box) : \tau_1 \times \tau_2 \Rightarrow \tau_2}
$$

Functions.

$$
\begin{array}{lcl}
k > \texttt{fn}(\tau, x.e) & \mapsto_{\mathsf{c}} & k < \texttt{fn}(\tau, x.e) \\
\\
k > \texttt{apply}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{apply}(\Box, e_2) > e_1 \\
k \triangleright \texttt{apply}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \texttt{apply}(v_1, \Box) > e_2 \\
k \triangleright \texttt{apply}(v_1, \Box) < v_2 & \mapsto_{\mathsf{c}} & k > \{v_2/x\}e \\
 & & \quad (v_1 = \texttt{fn}(\tau, x.e))
\end{array}
$$

$$
\frac{\cdot \vdash e_2 : \tau_2}{\texttt{apply}(\Box, e_2) : (\tau_2 \to \tau_1) \Rightarrow \tau_1}
\qquad
\frac{\cdot \vdash v_1 : \tau_2 \to \tau_1 \quad v_1 \text{ value}}{\texttt{apply}(v_1, \Box) : \tau_2 \Rightarrow \tau_1}
$$

Recursion.

$$
k > \texttt{rec}(\tau, x.e) \quad \mapsto_{\mathsf{c}} \quad k > \{\texttt{rec}(\tau, x.e)/x\}e
$$

Conditionals.

$$
\begin{array}{lcl}
k > \texttt{true} & \mapsto_{\mathsf{c}} & k < \texttt{true} \\
k > \texttt{false} & \mapsto_{\mathsf{c}} & k < \texttt{false} \\
k > \texttt{if}(e, e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{if}(\Box, e_1, e_2) > e \\
k \triangleright \texttt{if}(\Box, e_1, e_2) < \texttt{true} & \mapsto_{\mathsf{c}} & k > e_1 \\
k \triangleright \texttt{if}(\Box, e_1, e_2) < \texttt{false} & \mapsto_{\mathsf{c}} & k > e_2
\end{array}
$$

$$
\frac{\cdot \vdash e_1 : \tau \quad \cdot \vdash e_2 : \tau}{\texttt{if}(\Box, e_1, e_2) : \texttt{bool} \Rightarrow \tau}
$$

As an example, consider the evaluation of

```
(fn x:int => x) 0
```

$$
\begin{array}{lll}
\bullet & > & \mathtt{apply}(\mathtt{fn}(\mathtt{int}, x.x), \mathtt{num}(0)) \\
\mapsto_{\mathsf{c}} \bullet \triangleright \mathtt{apply}(\Box, \mathtt{num}(0)) & > & \mathtt{fn}(\mathtt{int}, x.x) \\
\mapsto_{\mathsf{c}} \bullet \triangleright \mathtt{apply}(\Box, \mathtt{num}(0)) & < & \mathtt{fn}(\mathtt{int}, x.x) \\
\mapsto_{\mathsf{c}} \bullet \triangleright \mathtt{apply}(\mathtt{fn}(\mathtt{int}, x.x), \Box) & > & \mathtt{num}(0) \\
\mapsto_{\mathsf{c}} \bullet \triangleright \mathtt{apply}(\mathtt{fn}(\mathtt{int}, x.x), \Box) & < & \mathtt{num}(0) \\
\mapsto_{\mathsf{c}} \bullet & > & \mathtt{num}(0) \\
\mapsto_{\mathsf{c}} \bullet & < & \mathtt{num}(0)
\end{array}
$$

Note that in the second-to-last step, $\{\mathtt{num}(0)/x\}x = \mathtt{num}(0)$

Before talking about the correctness of the C-machine, we state the progress and preservation theorems we expect. We do not prove these properties here, since they introduce no new techniques. Critical for progress is once again the value inversion lemma, as it is for the structural operational semantics.

Theorem 1 (Preservation and Progress for C-Machine)

(i) (Preservation) If $s : \sigma$ *and* $s \mapsto_{\mathsf{c}} s'$ *then* $s' : \sigma$.

(ii) (Progress) If $s : \sigma$ *then either*

(a) $s = (\bullet < v)$ *for some value* v*, or*

(b) $s \mapsto_{\mathsf{c}} s'$ *for some state* s'.

Proving the correctness of the C-machine is complicated by the fact that the two machines step at different rates. We further have to account for the stack. However, in the overall statement of the correctness theorem, these problems may not be apparent. In order to state the theorem, we first define the multi-step versions of the two transition judgments. This is just the reflexive and transitive closure of the single-step relation. We only define this formally for the abstract machine; other transition relations can similarly be extended to multiple steps [Ch. 2].

$s \mapsto_{\mathsf{c}}^* s'$ s steps to s' in zero or more steps

$$\frac{}{s \mapsto_{\mathsf{c}}^* s}\ \text{refl} \qquad \frac{s \mapsto_{\mathsf{c}} s' \quad s' \mapsto_{\mathsf{c}}^* s''}{s \mapsto_{\mathsf{c}}^* s''}\ \text{step}$$

We take certain elementary properties of the multi-step transition relation for granted and use them tacitly. We give here only one, as an example.

Theorem 2 (Transitivity)
If $s \mapsto_{\mathsf{c}}^* s'$ *and* $s' \mapsto_{\mathsf{c}}^* s''$ *then* $s \mapsto_{\mathsf{c}}^* s''$.

Proof: By straightforward rule induction on the derivation of $s \mapsto_{\mathsf{c}}^* s'$. ■

Theorem 3 (Correctness of C-Machine)
$e \mapsto^* v$ *if and only if* $\bullet > e \mapsto_{\mathsf{c}}^* \bullet < v$

As usual, we cannot prove this directly, but we need to generalize it. In this case we also need two lemmas.

Lemma 4 (Determinism)
If $s \mapsto_{\mathsf{c}} s'$ *and* $s \mapsto_{\mathsf{c}} s''$ *then* $s' = s''$.

Proof: By cases on the two given judgments. This is a degenerate case of rule induction, since the $\mapsto_{\mathsf{c}}$ judgment is defined only by axioms. ■

Lemma 5 (Value Computation)
(i) $k > v \mapsto_{\mathsf{c}}^* k < v$

(ii) If $k > v \mapsto_{\mathsf{c}}^* \bullet < a$ *then the computation decomposes into* $k > v \mapsto_{\mathsf{c}}^* k < v$ *and* $k < v \mapsto_{\mathsf{c}}^* \bullet < a$

Proof: Part (i) follows by induction on the structure of v.[1] Part (ii) then follows from part (i) by determinism. We show the proof of part (i) in detail.

Cases: $v = \mathtt{num}(n)$, $v = \mathtt{true}$, $v = \mathtt{false}$, or $v = \mathtt{fn}(\tau, x.e)$. Then the result is immediate by a single step of the abstract machine.

Case: $v = \mathtt{pair}(v_1, v_2)$. Then

$k > \mathtt{pair}(v_1, v_2)$
$\mapsto_{\mathsf{c}} k \triangleright \mathtt{pair}(\Box, v_2) > v_1$ By rule
$\mapsto_{\mathsf{c}}^* k \triangleright \mathtt{pair}(\Box, v_2) < v_1$ By i.h. on v_1
$\mapsto_{\mathsf{c}} k \triangleright \mathtt{pair}(v_1, \Box) > v_2$ By rule
$\mapsto_{\mathsf{c}}^* k \triangleright \mathtt{pair}(v_1, \Box) < v_2$ By i.h. on v_2
$\mapsto_{\mathsf{c}} k < \mathtt{pair}(v_1, v_2)$ By rule

[1] Equivalently, we could say: *By rule induction on the derivation of* v value.

■

Now we are in a position to prove the generalization that directly relates a single step in the original semantics to possibly several steps in the C-machine. The easiest way to arrive at the particular generalization we have below it to try to prove our overal theorem directly and then allow for a general stack k (instead of forcing the empty stack $\bullet$). Looking ahead at how this (and the value computation) lemma are used in the proof of Theorem 7 is quite instructive.

We express that if $e \mapsto e'$, then under any stack k, if the evaluation of e' yields the final answer a, then the evaluation of e also yields the final answer a.

Lemma 6 (Completeness Lemma for the C-Machine)
If $e \mapsto e'$ and $k > e' \mapsto_{\mathsf{c}}^* \bullet < a$ then $k > e \mapsto_{\mathsf{c}}^* \bullet < a$.

Proof: The proof is by rule induction on the derivation of $e \mapsto e'$.

Below, when we claim a step follow "*by inversion*" it is because exactly one of the rules could be applied as the first step. Technically, this is an inversion on the definition of $\mapsto_{\mathsf{c}}^*$ (rule step must have been applied), followed by an second inversion on the (single) first step that could have been taken.

We show only the cases for products, since all other cases follow a similar pattern.

For the search rules, we apply inversion until we have uncovered a subcomputation of the abstract machine to which we can apply the induction hypothesis. Then we reconstitute the full computation.

For the reduction rules, we directly construct the needed computation, possibly applying to the value computation lemma, part (i).

Case:

$$\frac{e_1 \mapsto e_1'}{\mathtt{pair}(e_1, e_2) \mapsto \mathtt{pair}(e_1', e_2)}$$

$e_1 \mapsto e_1'$		Subderivation
$k > \mathtt{pair}(e_1', e_2)$	$\mapsto_{\mathsf{c}}^* \bullet < a$	Assumption
$k > \mathtt{pair}(e_1', e_2) \mapsto_{\mathsf{c}} k \triangleright \mathtt{pair}(\Box, e_2) > e_1' \mapsto_{\mathsf{c}}^* \bullet < a$		By inversion
	$k \triangleright \mathtt{pair}(\Box, e_2) > e_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By i.h.
$k > \mathtt{pair}(e_1, e_2) \mapsto_{\mathsf{c}} k \triangleright \mathtt{pair}(\Box, e_2) > e_1 \mapsto_{\mathsf{c}}^* \bullet < a$		By rule

Case:

$$\frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{pair}(v_1, e_2) \mapsto \texttt{pair}(v_1, e_2')}$$

$e_1 \mapsto e_1'$	Subderivation
$k > \texttt{pair}(v_1, e_2') \mapsto_{\mathsf{c}}^* \bullet < a$	Assumption
$k > \texttt{pair}(v_1, e_2') \mapsto_{\mathsf{c}} k \triangleright \texttt{pair}(\Box, e_2') > v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By inversion
$k \triangleright \texttt{pair}(\Box, e_2') > v_1 \mapsto_{\mathsf{c}}^* k \triangleright \texttt{pair}(\Box, e_2') < v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By value computation (ii)
$k \triangleright \texttt{pair}(\Box, e_2') < v_1 \mapsto_{\mathsf{c}} k \triangleright \texttt{pair}(v_1, \Box) > e_2' \mapsto_{\mathsf{c}}^* \bullet < a$	By inversion
$k \triangleright \texttt{pair}(v_1, \Box) > e_2 \mapsto_{\mathsf{c}}^* \bullet < a$	By i.h.
$k \triangleright \texttt{pair}(\Box, e_2) < v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By rule
$k \triangleright \texttt{pair}(\Box, e_2) > v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By value computation (i)
$k > \texttt{pair}(v_1, e_2) \mapsto_{\mathsf{c}} k \triangleright \texttt{pair}(\Box, e_2) > v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By rule

Case:

$$\frac{e_1 \mapsto e_1'}{\texttt{fst}(e_1) \mapsto \texttt{fst}(e_1')}$$

$e_1 \mapsto e_1'$	Subderivation
$k > \texttt{fst}(e_1') \mapsto_{\mathsf{c}}^* \bullet < a$	Assumption
$k > \texttt{fst}(e_1') \mapsto_{\mathsf{c}} k \triangleright \texttt{fst}(\Box) > e_1' \mapsto_{\mathsf{c}}^* \bullet < a$	By inversion
$k \triangleright \texttt{fst}(\Box) > e_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By i.h.
$k > \texttt{fst}(e_1) \mapsto_{\mathsf{c}} k \triangleright \texttt{fst}(\Box) > e_1 \mapsto_{\mathsf{c}}^* \bullet < a$	By rule

Case:

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{fst}(\texttt{pair}(v_1, v_2)) \mapsto v_1}$$

$k < v_1 \mapsto_{\mathsf{c}}^* \bullet < a$	Assumption
$k > \texttt{fst}(\texttt{pair}(v_1, v_2))$	
$\mapsto_{\mathsf{c}} k \triangleright \texttt{fst}(\Box) > \texttt{pair}(v_1, v_2)$	By rule
$\mapsto_{\mathsf{c}}^* k \triangleright \texttt{fst}(\Box) < \texttt{pair}(v_1, v_2)$	By value computation (i)
$\mapsto_{\mathsf{c}} k < v_1$	By rule
$\mapsto_{\mathsf{c}}^* \bullet < a$	By assumption

Case:

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\mathtt{snd}(\mathtt{pair}(v_1, v_2)) \mapsto v_2}$$

$k < v_2 \mapsto_{\mathsf{c}}^* \bullet < a$	Assumption
$k > \mathtt{snd}(\mathtt{pair}(v_1, v_2))$	
$\mapsto_{\mathsf{c}} k \triangleright \mathtt{snd}(\Box) > \mathtt{pair}(v_1, v_2)$	By rule
$\mapsto_{\mathsf{c}}^* k \triangleright \mathtt{snd}(\Box) < \mathtt{pair}(v_1, v_2)$	By value computation (i)
$\mapsto_{\mathsf{c}} k < v_2$	By rule
$\mapsto_{\mathsf{c}}^* \bullet < a$	By assumption

■

We do not show the proof in the other direction, which is a minor variant of the one in [Ch. 11.1]. We now return to the overall correctness theorem.

Theorem 7 (Correctness of C-Machine)

(i) If $e \mapsto^* v$ *then* $\bullet > e \mapsto_{\mathsf{c}}^* \bullet < v$.

(ii) If $\bullet > e \mapsto_{\mathsf{c}}^* \bullet < v$ *then* $e \mapsto^* v$.

Proof: We show part (i) and omit part (ii) (see [Ch. 11.1]). The proof of part (i) is by induction on the derivation of $e \mapsto^* v$.

Case:

$$\frac{}{v \mapsto^* v} \text{ refl}$$

$\bullet > v \mapsto_{\mathsf{c}}^* \bullet < v$	By value computation (i)

Case:

$$\frac{e \mapsto e' \quad e' \mapsto^* v}{e \mapsto^* v} \text{ step}$$

$\bullet > e' \mapsto_{\mathsf{c}}^* \bullet < v$	By i.h.
$\bullet > e \mapsto_{\mathsf{c}}^* \bullet < v$	By completeness lemma

■

Supplementary Notes on Continuations

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 10
September 30, 2004

In this lecture we first introduce *exceptions* [Ch. 13] and then *continuations* [Ch. 12], two advanced control construct available in some functional languages.

Exceptions are a standard construct in ML and other languages such as Java. We give here only a particularly simple form; a more elaborate form is pursued in Assignment 4. It is particularly easy to describe now that our abstract machines makes a control stack explicit.

We introduce a new form of state

$$k \ll \mathsf{fail}$$

which signals that we are propagating an exception upwards in the control stack k, looking for a handler or stopping at the empty stack. This "uncaught exception" is a particularly common form of implementing runtime errors. We do not distinguish different exceptions, only failure.

We have two new forms of expressions $\mathtt{fail}$[1] and $\mathtt{try}(e_1, e_2)$ (with concrete syntax $\mathtt{try}\, e_1\, \mathtt{ow}\, e_2$). Informally, $\mathtt{try}(e_1, e_2)$ evaluates e_1 and returns its value. If the evaluation of e_1 fails, that is, an exception is raised, then we evaluate e_2 instead and returns its value (or propagate *its* exception). These rules are formalized in the C-machine as follows.

$$\begin{array}{lcll}
k > \mathtt{try}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \mathtt{try}(\Box, e_2) > e_1 & \\
k \triangleright \mathtt{try}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k < v_1 & \\
k > \mathtt{fail} & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} & \\
k \triangleright f \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} & \text{for } f \neq \mathtt{try}(\Box, _) \\
k \triangleright \mathtt{try}(\Box, e_2) \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k > e_2 &
\end{array}$$

[1] A type should be included here in order to preserve the property that every well-typed expression has a unique type, but we prefer not to complicate the syntax at this point.

In order to verify that these rules are sensible, we should prove appropriate progress and preservation theorems. In order to do this, we need to introduce some typing judgments for machine states and the new forms of expressions. First, expressions:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}(e_1, e_1) : \tau}$$

We can now state (without proof) the preservation and progress properties. The proofs follow previous patterns (see [Ch. 13]).

1. (Preservation) If $s : \sigma$ and $s \mapsto s'$ then $s' : \sigma$.
2. (Progress) If $s : \sigma$ then either
   (i) $s \mapsto s'$ for some s', or
   (ii) $s = \bullet < v$ with v value, or
   (iii) $s = \bullet \ll$ fail.

The manner in which the C-machine operates with respect to exceptions may seem a bit unrealistic, since the stack is unwound frame by frame. However, in languages like Java this is not an unusual implementation method. In ML, there is more frequently a second stack containing only handlers for exceptions. The handler at the top of the stack is innermost and a `fail` expression can jump to it directly.

Overall, such a machine should be equivalent to the specification of exceptions above, but potentially more efficient. Often, we want to describe several aspects of execution behavior of a language constructs in several different machines, keeping the first as high-level as possible. However, we will not pursue this further, but move on to the discussion of continuations. Continuations are more flexible, but also more dangerous than exceptions.

Continuations are part of the definition of Scheme and are implemented as a library in Standard ML of New Jersey, even though they are not part of the definition of Standard ML. Continuations have been described as the `goto` of functional languages, since they allow non-local transfer of control. While they are powerful, programs that exploit continuations can difficult to reason about and their gratuitous use should therefore be avoided.

There are two basic constructs, given here with concrete and abstract syntax. We ignore issues of type-checking in the concrete syntax.[2]

[2] See Assignment 4 for details on concrete syntax.

`callcc` x `=>` e	`callcc`$(x.e)$
`throw` e_1 `to` e_2	`throw`(e_1, e_2)

In brief, `callcc` x `=>` e captures the stack (= continuation) k in effect at the time the `callcc` is executed and substitutes `cont`(k) for x in e. we can later transfer control to k by throwing a value v to k with `throw` v `to` `cont`(k). Note that the stack k we capture can be returned past the point in which it was in effect. As a result, `throw` can effect a kind of "time travel". While this can lead to programs that are very difficult to understand, it has multiple legitimate uses. One pattern of usage is as an alternative to exceptions, another is to implement co-routines or threads. Another use is to achieve backtracking.

As a starting example we consider simple arithmetic expressions.

(a) `1 + callcc x => 2 + (throw 3 to x)` $\mapsto_{\mathsf{c}}^{*} 4$
(b) `1 + callcc x => 2` $\mapsto_{\mathsf{c}}^{*} 3$
(c) `1 + callcc x => if (throw 2 to x) then 3 else 4 fi`
$\mapsto_{\mathsf{c}}^{*} 3$

Example (a) shows an upward use of continuations similar to exceptions, where the addition of $2 + \Box$ is bypassed and discarded when we throw to x.

Example (b) illustrates that captured continuations need not be used in which case the normal control flow remains in effect.

Example (c) demonstrates that a `throw` expression can occur anywhere; its type does not need to be tied to the type of the surrounding expression. This is because a `throw` expression never returns normally—it always passes control to its continuation argument.

With this intuition we can describe the operational semantics, followed by the typing rules.

$$
\begin{array}{lcl}
k > \texttt{callcc}(x.e) & \mapsto_{\mathsf{c}} & k > \{\texttt{cont}(k)/x\}e \\
k > \texttt{throw}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{throw}(\Box, e_2) > e_1 \\
k \triangleright \texttt{throw}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \texttt{throw}(v_1, \Box) > e_2 \\
k \triangleright \texttt{throw}(v_1, \Box) < \texttt{cont}(k_2) & \mapsto_{\mathsf{c}} & k_2 < v_1 \\
k > \texttt{cont}(k') & \mapsto_{\mathsf{c}} & k < \texttt{cont}(k')
\end{array}
$$

The typing rules can be derived from the need to make sure both preservation and progress to hold. First, the constructs that can appear in the source.

$$\frac{\Gamma, x{:}\tau \text{ cont} \vdash e : \tau}{\Gamma \vdash \texttt{callcc}(x.e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \text{ cont}}{\Gamma \vdash \texttt{throw}(e_1, e_2) : \tau}$$

Finally, the rules for continuation values that can only arise during computation. They are needed to check the machine state, even though they are not needed to type-check the input.

$$\frac{k : \tau \Rightarrow \sigma}{\Gamma \vdash \texttt{cont}(k) : \tau \text{ cont}}$$

It looks like there could be a problem here, because σ, the final answer type of the continuation, does not appear in the conclusion. Fortunately, it works, but only because the final answer type σ of all continuations that may occur in a computation will be equal. To be precise, if we want to be talk about typing intermediate states of the computation, we would need to pass along the final answer type σ through the typing judgments.

As a more advanced example, consider the problem of composing a function with a continuation. This can also be viewed as explicitly pushing a frame onto a stack, represented by a continuation. Even though we have not yet discussed polymorphism, we will phrase it as a generic problem:

> Write a function
>
> ```
> compose : ('a -> 'b) -> 'b cont -> 'a cont
> ```
>
> so that `compose` F K returns a continuation K_1. Throwing a value v to K_1 should first compute $F\,v$ and then throw the resulting value v' to K.

To understand the solution, we analyze the intended behavior of K_1. When given a value v, it first applies F to v. So

$$K_1 = K_2 \triangleright \texttt{apply}(F, \Box)$$

for some K_2. Then, it needs to throw the result to K. So

$$K_2 = K_3 \triangleright \texttt{throw}(\Box, K)$$

and therefore

$$K_1 = K_3 \triangleright \texttt{throw}(\Box, K) \triangleright \texttt{apply}(F, \Box)$$

for some K_3.

How can we create such a continuation? The expression

```
throw (F ...) to K
```

will create a continuation of the form above. This continuation will be the stack precisely when the hole "..." is reached. So we need to capture it there:

```
throw (F (callcc k1 => ...)) to K
```

The next conundrum is how to return `k1` as the result of the `compose` function, now that we have captured it. Certainly, we can *not* just replace ... by `k1`, because the F would be applied (which is not only wrong, but also not type-correct). Instead we have to throw `k1` out of the local context! In order to throw it to the right place, we have to name the continuation in effect when the `compose` is called.

```
callcc r =>
   throw (F (callcc k1 => throw k1 to r)) to K
```

Now it only remains to abstract over F and K, where we take the liberty of writing a curried function directly in our language.

```
fun compose (f:'a -> 'b) (k:'b cont) : 'a cont is
  callcc r =>
     throw (f (callcc k1 => throw k1 to r)) to k
end
```

In order to verify the correctness of this function, we can just calculate, using the operational semantics, what happens when `compose` is applied to two values F and K under some stack K_0. This is a very useful exercise, because the correctness of many opaque functions can be verified in this way (and many incorrect functions discovered).

$$
\begin{array}{ll}
 & K_0 > \texttt{apply}(\texttt{apply}(\mathsf{compose}, F), K) \\
\mapsto_{\mathsf{c}}^* & K_0 > \texttt{callcc}(r.\texttt{throw}(\_, \texttt{apply}(F, \texttt{callcc}(k_1.\texttt{throw}(\_, k_1, r))), K)) \\
\mapsto_{\mathsf{c}} & K_0 > \texttt{throw}(\_, \texttt{apply}(F, \texttt{callcc}(k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0)))), K) \\
\mapsto_{\mathsf{c}} & K_0 \triangleright \texttt{throw}(\_, \square, K) > \texttt{apply}(F, \texttt{callcc}(k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0)))) \\
\mapsto_{\mathsf{c}}^* & K_0 \triangleright \texttt{throw}(\_, \square, K) \triangleright \texttt{apply}(F, \square) > \texttt{callcc}(k_1.\texttt{throw}(\_, k_1, \texttt{cont}(K_0)))
\end{array}
$$

At this point, we define

$$K_1 = K_0 \triangleright \texttt{throw}(\_, \square, K) \triangleright \texttt{apply}(F, \square)$$

and continue

$$
\begin{array}{ll}
\mapsto_{\mathsf{c}} & K_1 > \texttt{throw}(\_, K_1, \texttt{cont}(K_0)) \\
\mapsto_{\mathsf{c}} & K_0 < K_1
\end{array}
$$

By looking at K_1 we can see hat it exactly satisfies our specification. Interestingly, K_3 from our earlier motivation turns out to be K_0, the continuation in effect at the evaluation of compose. Note that if F terminates normally, then that part of the continuation is discarded because K is installed instead as specified. However, if F raises an exception, control is returned back to the point where the `compose` was called, rather than to the place where the resulting continuation was invoked (at least in our semantics). This is an example of the rather unpleasant interactions that can take place between exceptions and continuations.

See the code[3] for a rendering of this in Standard ML of New Jersey, where we have slightly different primitives. The translations are as given below. Note that, in particular, the arguments to `throw` are reversed which may be significant in some circumstances because of the left-to-right evaluation order.

Concrete MinML	Abstract MinML	SML of NJ
`callcc` x `=>` e	`callcc`$(x.e)$	`callcc (fn x => e)`
`throw` e_1 `to` e_2	`throw`(e_1, e_2)	`throw e2 e1`

For a simpler and quite practical example for the use of continuation refer to the implementation of threads given in the textbook [Ch. 12.3]. A runnable version of this code can be found at the same location as the example above.

[3] `http://www.cs.cmu.edu/~fp/courses/312/code/10-continuations/`

Supplementary Notes on Parametric Polymorphism

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 11
September 30, 2003

After an excursion into advanced control constructs, we return to the basic questions of type systems in the next couple of lectures. The first one addresses a weakness of the language we have presented so far: every expression has exactly one type. Some functions (such as the identity function `fn x => x`) should clearly be applicable at more than one type. We call such function *polymorphic*. We later distinguish two principal forms of polymorphism, namely *parameteric* and *ad hoc* polymorphism.

Briefly, a polymorphic construct is said to be *parametric* if it behaves the same at all its types. The identity function is an example of a function that is parametric in this sense. A function such as addition also has more than one type, at least `+ : int * int -> int` and `+ : float * float -> float`, but the function behaves differently at these two types: one implementation manipulates floating point representations the other integers.

Besides pure functions, there are many data structure (such as lists) whose element types should be arbitrary. We achieved this so far by making lists *primitive* in the language, but this trick does not extend when we try to write interesting programs over lists. For example, the following `map` function is clearly too specalized.

```
rec map : (int -> bool) -> int list -> bool list =>
  fn f : int -> bool =>
    fn l : int list =>
      case l
        of nil => nil[bool]
         | cons(x,l') => cons(f(x),map f l')
```

It should work for any $f : \tau \to \sigma$, $l : \tau$ `list` and return a result of type σ `list`. The importance of this kind of generic programming varies from language to language and application to application. It has always been considered central in functional programming in order to avoid unnecessary code duplication. In object-oriented programming it does not appear as critical, because subtyping and the class hierarchy allow some form of polymorphic programming. Nonetheless, the Java language has recently decided to add "generics" to its next revision—we will discuss later how this relates to parametric polymorphism as we present it here.

There are different ways to approach polymorphism. In its *intrinsic* form we allow polymorphic functions, but we are careful to engineer the language so that every function still has a unique type. This may sound contradictory, but it is in fact possible with a suitable extension of the expression language. In its *extrinsic* form, we allow an expresson to have multiple types, but we ensure that there is a *principal type* that subsumes (in a suitable sense) all other types an expression might have. The polymorphism of ML is extrinsic; nonetheless, we present it in its intrinsic form first.

The idea is to think of the `map` function above not only takes f and l as arguments, but also the type τ and σ. Fortunately, this does not mean we actually have to pass them at run-time, as we discuss later. We write `Fn` t `=>` e for a function that take a *type* as an argument. The (bound) type variable t stands for that argument in the body, e. The type of such a function is written a $\forall t.\tau$, where τ is the type of the body. To apply a function e to a type argument τ (called *instantiation*), we write e `[`τ`]`. We also introduce a short, mathematical notation for functions that are not recursive, called λ-abstraction.

Concrete	Abstract	Mathematical
`All t.` τ	`All`$(t.\tau)$	$\forall t.\tau$
`Fn` t `=>` e	`Fn`$(t.e)$	$\Lambda t.e$
e `[`τ`]`	`Inst`(e, τ)	$e[\tau]$
`fn` x`:`τ `=>` e	`fn`$(\tau, x.e)$	$\lambda x{:}\tau.\ e$

Using this notation, we can rewrite the example above.

```
Fn t => Fn s =>
rec map : (t -> s) -> t list -> s list =>
  fn f : t -> s =>
    fn l : t list =>
      case l
        of nil => nil[s]
         | cons(x,l') => cons(f(x),map f l')
```

In order to formalize the typing rules, recall the judgment τ type. So far, this judgment was quite straightforward, with rules such as

$$\frac{\tau_1 \text{ type} \quad \tau_2 \text{ type}}{\texttt{arrow}(\tau_1, \tau_2) \text{ type}} \qquad \frac{\tau \text{ type}}{\texttt{list}(\tau) \text{ type}} \qquad \frac{}{\texttt{int} \text{ type}}$$

Now, types may contain type variables. An example is the type of the identity function, which is $\forall t.t \to t$, or the type of the `map` function, which is $\forall t.\forall s.(t \to s) \to \texttt{list}(t) \to \texttt{list}(s)$. So the typing judgment becomes *hypothetical*, that is, we may reason from assumption t type for variables t. In all the rules above, they are simply propagated (we show the example of the function type). In addition, we have new rule for universal quantification.

$$\frac{\tau_1 \text{ type} \quad \tau_2 \text{ type}}{\texttt{arrow}(\tau_1, \tau_2) \text{ type}} \qquad \frac{\Gamma, t \text{ type} \vdash \tau \text{ type}}{\Gamma \vdash \texttt{All}(t.\tau) \text{ type}}$$

In addition, the notion of hypothetical judgments yields the rule for type variables

$$\frac{}{\Gamma_1, t \text{ type}, \Gamma_2 \vdash t \text{ type}}$$

and a substitution property.

Lemma 1 (Type Substitution in Types)
If $\Gamma_1 \vdash \tau$ type *and* Γ_1, t type$, \Gamma_2 \vdash \sigma$ type *then* $\Gamma_1, \{\tau/t\}\Gamma_2 \vdash \{\tau/t\}\sigma$ type.

This is the idea behind higher-order abstract syntax and hypothetical judgments, applied now to the language of types. Note that even though we wrote Γ above, only assumptions of the form t type will actually be relevant to the well-formedness of types.

Now we can present the typing rules proper.

$$\frac{\Gamma, t \text{ type} \vdash e : \sigma}{\Gamma \vdash \mathtt{Fn}(t.e) : \mathtt{All}(t.\sigma)}$$

$$\frac{\Gamma \vdash e : \mathtt{All}(t.\sigma) \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \mathtt{Inst}(e, \tau) : \{\tau/t\}\sigma}$$

Let us consider the example of the polymorphic identity function to understand the substitution taking place in the last rule. You should read this derivation bottom-up to understand the process of type-checking.

$$\begin{array}{rcl} t \text{ type}, x{:}t & \vdash & x : t \\ t \text{ type} & \vdash & \mathtt{fn}(t, x.x) : \mathtt{arrow}(t, t) \\ \cdot & \vdash & \mathtt{Fn}(t.\mathtt{fn}(t, x.x)) : \mathtt{All}(t.\mathtt{arrow}(t, t)) \end{array}$$

If we abbreviate the identity function by id then it must be instantiated by (apply to) a type before it can be applied to an expression argument.

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{int}] : \mathtt{int} \to \mathtt{int} \\ \cdot & \vdash & id\,[\mathtt{int}]\,3 : \mathtt{int} \end{array}$$

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{bool}] : \mathtt{bool} \to \mathtt{bool} \\ \cdot & \vdash & id\,[\mathtt{bool}]\,\mathtt{true} : \mathtt{bool} \end{array}$$

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\mathtt{int}] : \mathtt{int} \to \mathtt{int} \\ \cdot & \nvdash & id\,[\mathtt{int}]\,\mathtt{true} : \mathtt{int} \end{array}$$

Using mathematical notation:

$$\begin{array}{rcl} t \text{ type}, x{:}t & \vdash & x : t \\ t \text{ type} & \vdash & \lambda x{:}t.\ x : t \to t \\ \cdot & \vdash & \Lambda t.\ \lambda x{:}t.\ x : \forall t.t \to t \end{array}$$

As should be clear from these rules, assumptions of the form t type also must appear while typing expression, since expressions contain types. Therefore, we need a second substitution property:

Lemma 2 (Type Substitution in Expressions)
If $\Gamma_1 \vdash \tau$ type *and* Γ_1, t type, $\Gamma_2 \vdash e : \sigma$ *then* $\Gamma_1, \{\tau/t\}\Gamma_2 \vdash \{\tau/t\}e : \{\tau/t\}\sigma$.

Note that we must substitute into Γ_2, because the type variable t may occur in some declaration $x{:}\sigma$ in Γ_2.

In the operational semantics we have a choice on whether to declare a type abstraction Fn t => e to be a value, or to reduce e. Intuitively, the latter cannot get stuck because t is a *type variable* not an ordinary variable, and therefore is never needed in evaluation. Even though it seems consistent, we know if now language that supports such evaluation in the presence of free type variables. This decision yields the following rules:

$$\frac{}{\mathsf{Fn}(t.e) \ \text{value}}$$

$$\frac{}{\mathtt{Inst}(\mathsf{Fn}(t.e), \tau) \mapsto \{\tau/t\}e} \qquad \frac{e \mapsto e'}{\mathtt{Inst}(e, \tau) \mapsto \mathtt{Inst}(e', \tau)}$$

From this it is routine to prove the progress and preservation theorems. For preservation, we need the type substitution lemmas stated earlier in this lecture. For progress, we need a new value inversion property.

Lemma 3 (Polymorphic Value Inversion)
If $\cdot \vdash v : \mathtt{All}(t.\tau)$ *and* v value *then* $v = \mathsf{Fn}(t.e')$ *for some* e'.

Theorem 4 (Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the transition derivation for e. In the case of the reduction of a polymorphic function to a type argument, we need the type substitution property. ■

Theorem 5 (Progress)
If $\cdot \vdash e : \tau$ *then either*

(i) e value, *or*

(ii) $e \mapsto e'$ *for some* e'

Proof: By rule induction on the typing derivation for e. We need polymorphic value inversion to show that all cases for a type instantiation are covered. ■

In our language the polymorphism is *parametric*, which means that the operation of a polymorphic function is independent of the type that it is applied to. Formalizing this observation requires some advanced techniques that we will not discuss in this course.

This can be contrasted with *ad hoc* polymorphism, in which the function may compute differently at different types. For example, if the function $+$ is overloaded, so it has type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$ and also type $\texttt{float} \times \texttt{float} \rightarrow \texttt{float}$, then we need to have two different implementations of the function. Another example may be a `toString` function whose behavior depends on the type of the argument.

Parametric polymorphism can often be implemented in a way that avoids carrying types at run-time. This is important because we do not want polymorphic functions to be inherently less efficient than ordinary functions. ML has the property that all polymorphic functions are parametric with polymorphic equality as the only exception. Ignoring polymorphic equality, this means we can avoid carrying type information at run-time. In practice, some time information is usually retained in order to support garbage collection or some optimization. How to best implement polymorphic languages is still an area of active research.

ML-style polymorphism is not quite as general as the one described here. This is so that polymorphic type inference remains decidable and has *principal types*. See [Ch 20.2] for a further discussion. We will return to the issue of type inference later in this course.

Parametric polymorphism, even in the restricted form in which it is present in ML, can be dangerous when the language also has effects such as mutable references. The most straightforward rules for polymorphism in its extrinsic form (where expressions have multiple types) are

$$\frac{\Gamma, t\ \mathsf{type} \vdash e : \sigma \quad e\ \mathsf{value}}{\Gamma \vdash e : \mathtt{All}(t.\sigma)}$$

$$\frac{\Gamma \vdash e : \mathtt{All}(t.\sigma) \quad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash e : \{\tau/t\}\sigma}$$

The only change to the previous system is that we do not allow types in expressions, and that the expression e remains the same for type abstraction

and application. In this system, when the language includes effects, the generalization rule must be restricted to values or it will be unsound. The prototypical example is the following ML code:

```
let val r = ref (fn x => x)
in
  r := (fn x => x + 1);
  (!r) true
end
```

Even though `(fn x => x)` : $\alpha \rightarrow \alpha$ we can not conclude that r : $\forall\alpha.(\alpha \rightarrow \alpha)$ref. If that were allowed, both the assignment to r and the dereferencing of r would be well-typed, even though the code obviously violates types safety. In our rule above, the restriction of e to values would rule out this generalation (`ref (fn x => x)` is not a value).

Lecture Notes on Data Abstraction

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 12
October 7, 2004

One of the most important ideas in programming is *data abstraction*. It refers to the property that clients of library code cannot access the internal data structures of the library implementation. The implementation remains *abstract*. Data abstraction is inherently a static property, that is, a property that must be verified before the program is run. This is because during execution the internal data structures of the library are, of course, present and must be manipulated by the running code. Hence, data abstraction is very closely tied to type-checking [Ch. 21].

Modern languages, such as ML and Java, support data abstraction, although the degree to which it is supported (or how easy it is to achieve) varies. Lower-level languages such as C do not support data abstraction because various unsafe constructs can be exploited in order to expose representations. This can have the undesirable effect that authors of widely used library code cannot change their implementations because such a change would break client code. Furthermore, ill-behaved clients can change the representation of a data type, potentially breaking the internal invariants of the library. Even the presence of a well-documented application programmers interface (API) is not much help if it can be easily circumvented due to weaknesses in the programming language.

In ML, abstraction is supported primarily at the level of modules. This can be justified in two ways: first, data abstraction is mostly a question of program interfaces and therefore it arises naturally at the point where we have to consider program composition and modules. Second, the ML core language has been carefully designed so that no type information needs to be supplied by the programmer: full type inference is decidable. In the presence of data abstraction this no longer makes sense since, as we will

see, an implementation does not uniquely determine its interface.

So how is data abstraction enforced in ML? Consider the following skeletal signature, presenting a very simple interface to an implementation of queues containing only integers.

```
signature QUEUE =
sig
  type q
  val empty : q
  val enq : int * q -> q
  val deq : q -> q * int (* may raise Empty *)
end;
```

This signature declares a type `q` which is abstract (no implementation of `q` is given). It then presents three operations on elements of this type. An implementation of this interface is a structure that matches the signature. Here is an extremely inefficient one.

```
structure Q :> QUEUE =
struct
  type q = int list
  val empty = nil
  fun enq (x,l) = x::l
  fun deq l = deq' (rev l)
  and deq' (y::k) = (rev k, y)
    | deq' (nil) = raise Empty
end;
```

Note that we use *opaque ascription* `:> QUEUE`, which is Standard ML's way to guarantee data abstraction. No client can see the definition of the type `Q.q`. For example, the last line in the following example fails type-checking.

```
val q21 = Q.enq (2, Q.enq (1, Q.empty));
val (q2, 1) = Q.deq q21;
val _ = hd q21;   (* TYPE ERROR HERE *)
```

This is because `hd` can operate only on lists, while `q21` is only known to have type `Q.q`. The implementation of `Q.q` as `int list` is hidden from the type-checker in order to ensure data abstraction. This means we can replace `Q` with a more efficient implementation by a pair of lists,

```
structure Q :> QUEUE =
struct
  type q = int list * int list
  val empty = (nil, nil)
  fun enq (x, (back, front)) = (x::back, front)
  fun deq (back, x::front) = ((back, front), x)
    | deq (back as _::_, nil) = deq (nil, rev back)
    | deq (nil, nil) = raise Empty
end;
```

and any client code will continue to work (although it may now work much faster).

In order to avoid the complications of a full module system, we introduce *existential types* $\exists t.\tau$, where t is a bound type variable. t represents the abstract type and τ represents the type of the operations on t. Returning to the example, the signature

```
signature QUEUE =
sig
  type q
  val empty : q
  val enq : int * q -> q
  val deq : q -> q * int (* may raise Empty *)
end;
```

is represented by the type

$$\exists q.q \times (\texttt{int} \times q \to q) \times (q \to q \times \texttt{int}).$$

Except for the missing names `empty`, `enq`, and `deq`, this carries the same information as the signature.

A value of an existential type is a tuple whose first component is the implementation of the type, and the second component is an implementation of the operations on that type. We write this as $\texttt{pack}(\sigma, e)$. For the sake of brevity, we show only part of the example:

```
structure Q :> QUEUE =
struct
  type q = int list
  val empty = nil
  ...
end;
```

is represented as

$$\texttt{pack}(\texttt{int list}, \texttt{pair}(\texttt{nil}, \ldots)) : \exists q.q \times \cdots$$

In contrast, the second implementation

```
structure Q :> QUEUE =
struct
  type q = int list * int list
  val empty = (nil, nil)
  fun enq (x, (back, front)) = (x::back, front)
  fun deq (back, x::front) = ((back, front), x)
    | deq (back as _::_, nil) = deq (nil, rev back)
    | deq (nil, nil) = raise Empty
end;
```

looks like

$$\texttt{pack}(\texttt{int list} \times \texttt{int list}, \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots)) : \exists q.q \times \cdots$$

From these examples we can deduce the typing rules. First, existential types introduce a new bound type variable.

$$\frac{\Gamma, t \text{ type} \vdash \tau \text{ type}}{\Gamma \vdash \exists t.\tau \text{ type}}$$

Second, the package that implements an existential type requires that the operations on the type respect the definition of the type. This is modeled in the rule by substituting the implementation type for the type variable in the body of the existential.

$$\frac{\Gamma \vdash \sigma \text{ type} \quad \Gamma \vdash e : \{\sigma/t\}\tau}{\Gamma \vdash \texttt{pack}(\sigma, e) : \exists t.\tau}$$

For example, if we take the first implementation above, the first two lines below justify the third.

$$\frac{\begin{array}{l}\cdot \vdash \texttt{int list}\ \mathsf{type} \\ \cdot \vdash \texttt{pair}(\texttt{nil}, \ldots) : \texttt{int list} \times \cdots\end{array}}{\cdot \vdash \texttt{pack}(\texttt{int list}, \texttt{pair}(\texttt{nil}, \ldots)) : \exists q.q \times \cdots}$$

In the second implementation, we need the implementation of q to have type $\texttt{int list} \times \texttt{int list}$.

$$\frac{\begin{array}{l}\cdot \vdash \texttt{int list} \times \texttt{int list}\ \mathsf{type} \\ \cdot \vdash \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots) : (\texttt{int list} \times \texttt{int list}) \times \cdots\end{array}}{\cdot \vdash \texttt{pack}(\texttt{int list} \times \texttt{int list}, \texttt{pair}(\texttt{pair}(\texttt{nil}, \texttt{nil}), \ldots)) : \exists q.q \times \cdots}$$

Next we have to consider how make the implementation of an abstract type available. In ML, a structure is available when a definition `structure S = ...` is made at the top level. Here, we need an explicit construct to open a package to make it available to a client. Given a package $e : \exists t.\tau$, we write $\texttt{open}(e, t.x.e')$ to make e available to the client e'. Here, t is a bound type variable that refers to the abstract type (and remains abstract in e') and x is a bound variable that stands for the implementation of the operations on the type. In our example, $\texttt{fst}(e)$ denotes the implementation of `empty`, $\texttt{fst}(\texttt{snd}(e))$ stands for the implementation of `enq`, etc.

This leads us to the following rule:

$$\frac{\Gamma \vdash e : \exists t.\tau \quad \Gamma, t\ \mathsf{type}, x{:}\tau \vdash e' : \sigma \quad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \texttt{open}(e, t.x.e') : \sigma}$$

We have added the explicit premise that $\Gamma \vdash \sigma\ \mathsf{type}$ to emphasize that t must not occur already in Γ or σ: every time we open a package, or multiple package, we obtain a new type, different from all types already known. This *generativity* means that even multiple instances of the exact same structure are not recognized to have the same implementation type: any one of them could be replaced by another one without affecting the correctness of the client code.

The property of data abstraction can be seen in the rule above: the code e' can use the library code e, but during type-checking only a type variable t is visible, not the implementation type. This means the code in e' is *parametric* in t, which guarantees data abstraction.

The operational semantics is straightforward and does not add any new ideas to those previously discussed. This confirms that the importance

of data abstraction lies in compile-time type-checking, not in the runtime properties of the language.

$$\frac{e \mapsto e'}{\texttt{pack}(\tau, e) \mapsto \texttt{pack}(\tau, e')} \qquad \frac{v \ \text{value}}{\texttt{pack}(\tau, v) \ \text{value}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{open}(e_1, t.x.e_2) \mapsto \texttt{open}(e_1', t.x.e_2)}$$

$$\frac{v_1 \ \text{value}}{\texttt{open}(\texttt{pack}(\tau, v_1), t.x.e_2) \mapsto \{v_1/x\}\{\tau/t\}e_2}$$

Observe that before the evaluation of the body of an `open` expression, we substitute τ for t, making the abstract type concrete. However, we know that e_2 was type-checked without knowing τ, so this does not violate data abstraction.

The progress and preservation theorems do not introduce any new ideas. For the type substitution we need a type substitution property that was given in Lecture 11 on *Parametric Polymorphism*.

Combining parametric polymorphism and data abstraction, that is, universal and existential types can be interesting and fruitful. For example, assume we would like to allows queues to have elements of arbitrary type s. This would be specified as

$$\forall a.\exists q.q \times (a \times q \to q) \times (q \to q \times a).$$

For example, the implementation of a queue by a single list would then have the form

$$\texttt{Fn}(a.\texttt{pack}(a\ \texttt{list}, \langle \texttt{Inst}(\texttt{nil}, a), \ldots \rangle))$$

Note that the type

$$\exists q.\forall a.q \times (a \times q \to q) \times (q \to q \times a)$$

would be incorrect, because we cannot choose the implementation type for q before we know the type a.

As another example, assume we want to widen the interface to also export double-ended queues `q'` with some additional operations that we leave unspecified here. Then the type would have the form

$$\exists q.\exists q'.q \times q' \times \cdots.$$

The implementation would provide definitions for both q and q', as in

$$\texttt{pack}(\texttt{int}\ \texttt{list}, \texttt{pack}(\texttt{int}\ \texttt{list}, \ldots)).$$

Next we return to the question of type-checking. Consider[1]

$$\texttt{pack}(\texttt{int}, \texttt{pair}(\texttt{fn}(\texttt{int}, x.x + 1), \texttt{fn}(\texttt{int}, x.x - 1))).$$

This package has 16 different types; we show four of them here:

$$\begin{array}{l}
\exists t.(t \to t) \times (t \to t) \\
\exists t.(\texttt{int} \to t) \times (t \to \texttt{int}) \\
\exists t.(t \to \texttt{int}) \times (\texttt{int} \to t) \\
\exists t.(\texttt{int} \to \texttt{int}) \times (\texttt{int} \to \texttt{int})
\end{array}$$

While not all of these are meaningful, they are all different and the type-checker has no way of guessing which one the programmer may have meant. This is inherent: an implementation does not determine its interface. However, we can *check* an implementation against an interface, which is precisely what bi-directional type-checking achieves. We have not formally presented the technique in these notes and postpone its discussion for now.

[1] using infix notation for addition and subtraction

Lecture Notes on Recursive Types

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 13
October 12, 2004

In the last two lectures we have seen two critical concepts of programming languages: parametric polymorphism (modeled by universal types) and data abstraction (modeled by existential types). These provide quantification over types, but they do not allow us to define types recursively. Clearly, this is needed in a practical language. Common data structures such as lists or trees are defined inductively, which is a restricted case of general recursion in the definition of types [Ch. 19.3].

So far, we have considered how to add a particular recursive type, namely lists, to our language as a primitive by giving constructors (`nil` and `cons`), a discriminating destructor (`listcase`). For a realistic language, this approach is unsatisfactory because we would have to extend the language itself every time we needed a new data type. Instead we would like to have a uniform construct to define new recursive types as we need them. In ML, this is accomplished with the `datatype` construct. Here we use a somewhat lower-level primitive—we return to the question how this is related to ML at the end of this lecture.

As a first, simple *non-recursive* example, consider how we might implement a three-element type.

```
datatype Color = Red | Green | Blue;
```

Using the singleton type 1 (`unit`, in ML), we can define

$$
\begin{array}{lcl}
\mathit{Color} & = & 1 + (1 + 1) \\
\mathit{Red} : \mathit{Color} & = & \mathsf{inl}\,(\,) \\
\mathit{Green} : \mathit{Color} & = & \mathsf{inr}(\mathsf{inl}\,(\,)) \\
\mathit{Blue} : \mathit{Color} & = & \mathsf{inr}(\mathsf{inr}\,(\,)) \\
\mathit{ccase} & : & \forall s.\mathit{Color} \to (1 \to s) \to (1 \to s) \to (1 \to s) \to s \\
& = & \Lambda s.\lambda c.\lambda y_1.\lambda y_2.\lambda y_3. \\
& & \quad \mathsf{case}\ c \\
& & \qquad \mathsf{of}\ \mathsf{inl}(c_1) \Rightarrow y_1\, c_1 \\
& & \qquad\ \mid \mathsf{inr}(c_2) \Rightarrow \mathsf{case}\ c_2 \\
& & \qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}(z_2) \Rightarrow y_2\, z_2 \\
& & \qquad\qquad\qquad\ \mid \mathsf{inr}(z_3) \Rightarrow y_3\, z_3
\end{array}
$$

Recall the notation $\lambda x.e$ for a non-recursive function and $\Lambda t.e$ for a type abstraction. The ccase constructs invokes one of its arguments y_1, y_2, or y_3, depending on whether the argument c represents red, green, or blue.

If we try to apply the technique, for example, to represent natural numbers as they would be given in ML by

```
datatype Nat = Zero | Succ of Nat;
```

we would have

$$\mathit{Nat} = 1 + (1 + (1 + \cdots))$$

where

$$n : \mathit{Nat} = \underbrace{\mathsf{inr}(\ldots(\mathsf{inr}(}_{n \text{ times}}\mathsf{inl}\,(\,))))$$

In order to make this definition recursive instead of infinitary we would write

$$\mathit{Nat} \simeq 1 + \mathit{Nat}$$

where we leave the mathematical status of $\simeq$ purposely vague, but one should read $\tau \simeq \sigma$ as "τ *is isomorphic to* σ". Just as with the recursion at the level of expressions, it is more convenient to write this out as an explicit definition using a recursion operator.

$$\mathit{Nat} = \mu t.1 + t$$

We can unwind a recursive type $\mu t.\sigma$ to $\{\mu t.\sigma/t\}\sigma$ to obtain an isomorphic type.

$$\mathit{Nat} = \mu t.1 + t \simeq \{\mu t.1 + t/t\}1 + t = 1 + \mu t.1 + t = 1 + \mathit{Nat}$$

In order to obtain a reasonable system for type-checking, we have constructors and destructors for recursive types. They can be considered "witnesses" for the unrolling of a recursive type.

$$\frac{\Gamma \vdash e : \{\mu t.\tau/t\}\tau \quad \Gamma \vdash \mu t.\tau \text{ type}}{\Gamma \vdash \mathsf{roll}(e) : \mu t.\tau} \qquad \frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \mathsf{unroll}(e) : \{\mu t.\tau/t\}\tau}$$

The operational semantics and values are straightforward; the difficulty of recursive types lies entirely in the complexity of the substitution that takes place during the unrolling of a recursive type.

$$\frac{e \mapsto e'}{\mathsf{roll}(e) \mapsto \mathsf{roll}(e')} \qquad \frac{v \text{ value}}{\mathsf{roll}(v) \text{ value}}$$

$$\frac{e \mapsto e'}{\mathsf{unroll}(e) \mapsto \mathsf{unroll}(e')} \qquad \frac{v \text{ value}}{\mathsf{unroll}(\mathsf{roll}(v)) \mapsto v}$$

Now we can go back to the definition of specific recursive types, using natural numbers built from zero and successor as the first example.

$$\begin{array}{lcl}
\mathit{Nat} & = & \mu t.1 + t \\
\mathit{Zero} : \mathit{Nat} & = & \mathsf{roll}(\mathsf{inl}\,(\,)) \\
\mathit{Succ} : \mathit{Nat} \to \mathit{Nat} & = & \lambda x.\mathsf{roll}(\mathsf{inr}\,x) \\
\mathit{ncase} & : & \forall s.\mathit{Nat} \to (1 \to s) \to (\mathsf{nat} \to s) \to s \\
& = & \Lambda s.\lambda n.\lambda y_1.\lambda y_2. \\
& & \quad \mathsf{case}\ \mathsf{unroll}(n) \\
& & \quad\quad \mathsf{of}\ \mathsf{inl}(z_1) \Rightarrow y_1\, z_1 \\
& & \quad\quad\ \mid \mathsf{inr}(z_2) \Rightarrow y_2\, z_2
\end{array}$$

In the definition of ncase we see that $z_1 : 1$ and $z_2 : \mathit{Nat}$, so that y_2 is really applies to the predecessor of n, while y_1 is just applied to the unit element.

Polymorphic recursive types can be defined in a similar manner. As an example, we consider lists with elements of type r.

$$\begin{array}{lcl}
r\ \mathit{List} & = & \mu t.1 + r \times t \\
\mathit{Nil} & : & \forall r.r\ \mathit{List} \\
& = & \Lambda r.\mathsf{roll}(\mathsf{inl}\,(\,)) \\
\mathit{Cons} & : & \forall r.r \times r\ \mathit{List} \to r\ \mathit{List} \\
& = & \Lambda s.\lambda p.\mathsf{roll}(\mathsf{inr}\,p) \\
\mathit{lcase} & : & \forall s.\forall r.r\ \mathit{List} \to (1 \to s) \to (r \times r\ \mathit{List} \to s) \to s \\
& = & \Lambda s.\Lambda r.\lambda l.\lambda y_1.\lambda y_2. \\
& & \quad \mathsf{case}\ \mathsf{unroll}(l) \\
& & \quad\quad \mathsf{of}\ \mathsf{inl}(z_1) \Rightarrow y_1\, z_1 \\
& & \quad\quad\ \mid \mathsf{inr}(z_2) \Rightarrow y_2\, z_2
\end{array}$$

If we go back to the first example, it is easy to see that representation of data types does not quite match their use in ML. This is because we can see the complete implementation of the type, for example, $Color = 1 + (1 + 1)$. This leads to a loss of data abstraction and confusion between different data types. Consider another ML data type

```
Answer = Yes | No | Maybe;
```

This would also be represented by

$$Answer = 1 + (1 + 1)$$

which is the *same* as $Color$. Perhaps this does not seem problematic until we realize that `Yes` $=$ `Red`! This is obviously meaningless and creates incompatibilities, for example, if we decide to change the order of definition of the elements of the data types.

Fortunately, we already have the tool of data abstraction to avoid this kind of confusion. We therefore combine *recursive types* with *existential types* to model the `datatype` construct of ML. Using the first example,

```
datatype Color = Red | Green | Blue;
```

we would represent this

$$\exists c.c \times c \times c \times \forall s.c \to (1 \to s) \to (1 \to s) \to (1 \to s) \to s$$

The implementation will have the form

$$\texttt{pack}(1 + (1 + 1), \texttt{pair}(\mathsf{inl}\,(\,), \texttt{pair}(\mathsf{inr}(\mathsf{inl}\,(\,)), \ldots)))$$

Upon opening an implementation of this type we can give its components the usual names. With this strategy, $Color$ and $Answer$ can no longer be confused.

We close this section with a curiosity regarding recursive types. We can use them to type a simple, non-terminating expression that does *not* employ recursive functions! The pure λ-calculus version of this function is $(\lambda x.x\,x)\,(\lambda x.x\,x)$. Our example is just slight more complicated because of the need to roll and unroll recursive types.

We define

$$\begin{array}{lcl}
\omega & = & \mu t.t \to t \\
x{:}\omega & \vdash & \mathsf{unroll}(x) : \omega \to \omega \\
x{:}\omega & \vdash & \mathsf{unroll}(x)\, x : \omega \\
\cdot & \vdash & \lambda x.\mathsf{unroll}(x)\, x : \omega \to \omega \\
\cdot & \vdash & \mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x) : \omega \\
\cdot & \vdash & (\lambda x.\mathsf{unroll}(x)\, x)\, \mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x) : \omega
\end{array}$$

When we execute this term we obtain

$$\begin{array}{l}
(\lambda x.\mathsf{unroll}(x)\, x)\, \mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x) \\
\mapsto \mathsf{unroll}(\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x))\, (\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x)) \\
\mapsto (\lambda x.\mathsf{unroll}(x)\, x)\, (\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\, x))
\end{array}$$

so it reduces to itself in two steps.

While we will probably not prove this in this course, recursive types are necessary for such an example. For any other (pure) type construct we have introduced so far, all functions are terminating if we do not use recursion at the level of expressions.

Lecture Notes on Subtyping

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 14
October 19, 2004

Subtyping is a fundamental idea in the design of programming languages. It can allow us to write more concise and readable programs by eliminating the need to convert explicitly between elements of types. It can also be used to express more properties of programs. Finally, it is absolutely fundamental in object-oriented programming where the notion of subclass is closely tied to the notion of subtype.

Which subtyping relationships we want to integrate into a language depends on many factors. Besides some theoretical properties we want to satisfy, we also have to consider the pragmatics of type-checking and the operational semantics. In this lecture we are interested in isolating the fundamental principles that must underly various forms of subtyping. We will then see different instances of how these principles can be applied in practice.

We write $\tau \leq \sigma$ to express that τ is a subtype of σ. The fundamental principle of subtyping says:

> *If $\tau \leq \sigma$ then wherever a value of type σ is required, we can use a value of type τ instead.*

This can be refined into two more specific statements, depending on the form of subtyping used.

> **Subset Interpretation.** *If $\tau \leq \sigma$ then every value of type τ is also a value of type σ.*

As an example, consider both empty and non-empty lists as subtypes of the type of lists. This is because an empty list is clearly a list, and a non-empty lists is also a list. One can see that with a subset interpretation of subtyping one can track properties of values.

> **Coercion Interpretation.** *If $\tau \leq \sigma$ then every value of type τ can be converted (coerced) to a value of type σ in a unique way.*

As an example, consider integers as a subtype of floating point numbers. This interpretation is possible because there is a unique way we can convert an integer to a corresponding floating point representation (ignoring questions of size bounds). Therefore, coercive subtyping allows us to omit explicit calls to functions that perform the coercion. However, we have to be careful to guarantee the coerced value is unique, because otherwise the result of a computation may be ambiguous. For example, if we want to say that both integers and floating point numbers are also a subtype of strings, and the coercion yields the printed representation, we violate the uniqueness guarantee. This is because we can coerce `3` to `"3"` since int $\leq$ string or `3` to `3.0` and then to `"3.0"` using first int $\leq$ float and then float $\leq$ string. We call a language that satisfies the uniqueness property *coherent*; incoherent languages are poor from the design point of view and can lead to many practical problems. We therefore require the coherence from the start.

Note that both forms of subtyping satisfy the fundamental principle, but that the coercion interpretation is more difficult to achieve than subset interpretation, because we have to verify uniqueness of coercions. Because it is somewhat richer, we concentrate in this lecture on working out a concrete system of subtyping under the coercion interpretation.

First, some general laws that are independent of whether we choose a subset or coercion interpretation. The defining property of subtyping can be expressed in the calculus by the rule of *subsumption*.

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash e : \sigma}\ subsume$$

Secondly, we have reflexivity and transitivity of subtyping.

$$\frac{}{\tau \leq \tau}\ refl \qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ trans$$

Let us carefully justify these principles. Under the subset interpretation $\tau \leq \tau$ follows from $A \subseteq A$ for any set of values A. Transitivity follows from

the transitivity of the subset relation. Under the coercion interpretation, the identity function coerces from τ to τ for any τ. And we can validate transitivity by composition of functions.

To make the latter considerations concrete, we annotate the subtyping judgment with a coercion and we calculate this coercion in each case. We write $f : \tau \leq \sigma$ is f is a coercion from τ to σ. Note that coercions f are always closed, that is, contain no free variables, so no context is necessary.

$$\frac{}{\lambda x.x : \tau \leq \tau}\ refl \qquad \frac{f : \tau_1 \leq \tau_2 \quad g : \tau_2 \leq \tau_3}{\lambda x.g(f(x)) : \tau_1 \leq \tau_3}\ trans$$

The three laws we have are essentially all the general laws that can be formulated in this manner. Coherence is stated in a way that is similar to a substitution principle.

Coherence. *If $f : \tau \leq \sigma$ and $g : \tau \leq \sigma$ then $f \simeq g : \tau \to \sigma$.*

Here, extensional equality $f \simeq g : \tau$ is defined inductively on type τ. Note that all coercions should terminate and not have any effects. We show the cases for functions, pairs, and primitive types.

1. $e \simeq e' : \mathsf{int}$ if $e \mapsto^* \mathtt{num}(n)$ and $e' \mapsto^* \mathtt{num}(n')$ and $n = n'$.
2. $e \simeq e' : \tau_1 \times \tau_2$ if $\mathtt{fst}(e) \simeq \mathtt{fst}(e') : \tau_1$ and $\mathtt{snd}(e) \simeq \mathtt{snd}(e') : \tau_2$.
3. $e \simeq e' : \tau_1 \to \tau_2$ if for any $e_1 \simeq e_1' : \tau_1$ we have $e\, e_1 \simeq e'\, e_1' : \tau_2$.

As a particular example of subtyping, consider $\mathsf{int} \leq \mathsf{float}$. We call the particular coercion $\mathsf{itof} : \mathsf{int} \to \mathsf{float}$.

$$\frac{}{\mathsf{int} \leq \mathsf{float}} \qquad \frac{}{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}$$

In order to use these functions, consider two versions of the addition operation: one for integers and one for floating point numbers. We avoid overloading here, which is subject of another lecture.

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{float} \quad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 +. e_2 : \mathsf{float}}$$

Now an expression such as $2 + 3.0$ is ill-typed, since the second argument is a floating point number and floating point numbers in general cannot be coerced to integers. However, the expression $2 +. 3.0$ is well-typed

because the first argument 2 can be coerced to the floating point number 2.0 by applying itof. Concretely:

$$\dfrac{\dfrac{\overline{\vdash 2 : \mathsf{int}} \quad \overline{\mathsf{int} \leq \mathsf{float}}}{\vdash 2 : \mathsf{float}} \qquad \overline{\vdash 3.0 : \mathsf{float}}}{\vdash 2 +. 3.0 : \mathsf{float}}$$

So far we have avoided a discussion of the operational semantics, but we can see that (a) under the subset interpretation the operational semantics remains the same as without subtyping, and (b) under the coercion interpretation the operational semantics must apply the coercion functions. That is, we cannot define the operational semantics directly on expressions, because only the subtyping derivation will contain the necessary information on how and where to apply the coercions. We do not formalize the translation from subtyping derivations with coercions to the language without, but we show it by example. In the case above we have

$$\dfrac{\dfrac{\overline{\vdash 2 : \mathsf{int}} \quad \overline{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\vdash \mathsf{itof}(2) : \mathsf{float}} \qquad \overline{\vdash 3.0 : \mathsf{float}}}{\vdash \mathsf{itof}(2) +. 3.0 : \mathsf{float}}$$

The subsumption rule with annotations then looks like

$$\dfrac{\Gamma \vdash e : \tau \quad f : \tau \leq \sigma}{\Gamma \vdash f(e) : \sigma}$$

so we interpret $f : \tau \leq \sigma$ as $f : \tau \to \sigma$. However, typing derivations are not unique. As written, however, it is not quite right because the source code does not contain f, only the result of type checking. This process is generally called *elaboration* and will occupy us further in the next section. Writing out coercions as we did above, we could have

$$\dfrac{\dfrac{\overline{\vdash 2 : \mathsf{int}} \quad \dfrac{\overline{\lambda x.x : \mathsf{int} \leq \mathsf{int}} \quad \overline{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\lambda y.\mathsf{itof}((\lambda x.x)(y)) : \mathsf{int} \leq \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) : \mathsf{float}} \qquad \overline{\vdash 3.0 : \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) +. 3.0 : \mathsf{float}}$$

This alternative compilation will behave identically to the first one, itof and $\lambda y.\mathsf{itof}((\lambda x.x)(y))$ are observationally equivalent. To see this, apply both sides to a value v. Then the one side yiels $\mathsf{itof}(v)$, the other side

$$(\lambda y.\mathsf{itof}((\lambda x.x)(y)))v \mapsto \mathsf{itof}((\lambda x.x)v) \mapsto \mathsf{itof}(v)$$

The fact that the particular chosen typing derivation does not affect the behavior of the compiled expressions (where coercions are explicit) is the subject of the coherence theorem for a language. This is a more precise expression of the "uniqueness" required in the defining property for coercive subtyping.

At this point we have general laws for typing (subsumption) and subtyping (reflexivity and transitivity). But how does subtyping interact with pairs, functions, and other constructs? We start with pairs. We can coerce a value of type $\tau_1 \times \tau_2$ to a value of type $\sigma_1 \times \sigma_2$ if we can coerce the individual components appropriately. That is:

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}$$

With explicit coercions:

$$\frac{f_1 : \tau_1 \leq \sigma_1 \quad f_2 : \tau_2 \leq \sigma_2}{\lambda p.\mathtt{pair}(f_1(\mathtt{fst}(p)), f_2(\mathtt{snd}(p))) : \tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}$$

Functions are somewhat trickier. We know that $\mathsf{int} \leq \mathsf{float}$. It should therefore be clear that $\mathsf{int} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{float}$, because we can coerce the output of the function of the left to the required output for the function on the right. So

$$\lambda h.\lambda x.\mathsf{itof}(h(x)) : \mathsf{int} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{float}$$

Perhaps surprisingly, we have

$$\mathsf{float} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{int}$$

because we can obtain a function of the type on the right from a function on the left by coercing the argument:

$$\lambda h.\lambda x.h(\mathsf{itof}(x)) : \mathsf{float} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{int}$$

Putting these two ideas together we get

$$\lambda h.\lambda x.\mathsf{itof}(h(\mathsf{itof}(x))) : \mathsf{float} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{float}$$

In the general case, we obtain the following rule:

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

With coercion functions:

$$\frac{f_1 : \sigma_1 \leq \tau_1 \quad f_2 : \tau_2 \leq \sigma_2}{\lambda h.\lambda x.f_2(h(f_1(x))) : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

The fact that the subtyping relationship flips in the left premise is called *contravariance*. We say that function subtyping is contravariant in the argument and covariant in the result. Subtyping of pairs, on the other hand, is covariant in both components.

Mutable reference can be neither covariant nor contravariant. As simple counterexamples, consider the following pieces of code.

The first one assumes that τ ref $\leq \sigma$ ref if $\sigma \leq \tau$, that is reference subtyping is contravariant.

```
let val r = ref 2.1 (* r : float ref *)
in
   !r
end : int         (* using float ref <: int ref *)
```

Clearly, this is incorrect and violates preservation.

Conversely if we assume subtyping is covariant, that is, τ ref $\leq \sigma$ ref if $\tau \leq \sigma$, then

```
let val r = ref 3 (* r : int ref *)
in
   r := 2.1;    (* using int ref <: float ref *)
   !r
end : int
```

To avoid these counterexamples we make mutable references *non-variant*.

$$\frac{\tau \leq \sigma \quad \sigma \leq \tau}{\tau \text{ ref} \leq \sigma \text{ ref}}$$

More detailed analyses of references are possible. In particular, we can decompose them into "sources" from which we can only read and "sinks" to which we can only write. Sources are covariant and sinks are contravariant. Since we can both read from and write to mutable references, they must be non-variant. We will not develop this formally here.

Note that non-variance of references is an important issue in object-oriented languages. For example, in Java every element of an array acts

like a reference and should therefore be non-variant. However, in Java, arrays are co-variant, so run-time checks on types of assigments to arrays or mutable fields are necessary in order to save type preservation. In particular, every time one writes to an array of objects in Java, a dynamic tag-check is required, because arrays are co-variant in the element type. Fortunately, this is possible because in Java and other object-oriented languages there is enough information at run-time to perform this check reasonably efficiently.

There are other type constructors that must be non-variant. For example, if we define (in ML)

```
datatype 'a func = F of 'a -> 'a
```

then the rule

$$\frac{\tau \leq \sigma \quad \sigma \leq \tau}{\tau \text{ func} \leq \sigma \text{ func}}$$

is forced by the occurrence of `'a` in both a co-variant and contra-variant position in the argument to the constructor `F`.

Conversely, a type such as

```
datatype 'a singleton = Unit of unit
```

has only a single element, `Unit`, independently of the instantiation of `'a`. Therefore we can pose, for arbitrary τ and σ, that

$$\frac{}{\tau \text{ singleton} \leq \sigma \text{ singleton}}$$

without fear of compromising soundness.

Lecture Notes on Bidirectional Type Checking

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 17
October 21, 2004

At the beginning of this class we were quite careful to guarantee that every well-typed expression has a unique type. We relaxed our vigilance a bit when we came to constructs such as universal types, existential types, and recursive types, essentially because the question of unique typing became less obvious or, as in the case of existential types, impossible without excessive annotations.

In this lecture we first recall the notion of modes and mode correctness that allow us to interpret inference rules as an algorithm. We then apply this idea to develop an algorithm that propagates type information through an abstract syntax tree in two directions, allowing for a more natural type-checking algorithm we call *bidirectional*.

In either case, it is convenient to think of type checking as the process of bottom-up construction of a typing derivation. In that way, we can interpret a set of typing rules as describing an algorithm, although some restriction on the rules will be necessary (not every set of rules naturally describes an algorithm).

The idea behind modes is to label the constituents of a judgment as either *input* or *output*. For example, the typing judgment $\Gamma \vdash e : \tau$ should be such that Γ and e are input and τ is output (if it exists). We then have to check each rule to see if the annotations as input and output are consistent with a bottom-up reading of the rule. This proceeds as follows, assuming at first a single-premise inference rule. We refer to constituents of a judgment as either *known* or *free* during a particular stage of proof construction.

1. **Assume** each input constituent of the *conclusion* is known.

2. **Show** that each input constituent of the *premise* is known, and each output constituent of the premise is still free (unknown).
3. **Assume** that each output constituent of the *premise* is known.
4. **Show** that each output constituent of the *conclusion* is known.

Given the intuitive interpretation of an algorithm as proceeding by bottom-up proof construction, this method of checking should make some sense intuitively. As an example, consider the rule for functions.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) : \tau_1 \to \tau_2}\ \textit{FnTyp}$$

with the mode

$$\Gamma^+ \vdash e^+ : \tau^-$$

where we have marked inputs with + and outputs with -.

1. We **assume** that Γ, τ_1, and $x.e$ are known.
2. We **show** that $\Gamma, x{:}\tau_1$ and e are known and τ_2 is free, all of which follow from assumptions made in step 1.
3. We **assume** that τ_2 is also known.
4. We **show** that τ_1 and τ_2 are known, which follows from the assumptions made in steps 1 and 3.

Consequently our rule for function types is mode correct with respect to the given mode. If we had omitted the type τ_1 in the syntax for function abstraction, then the rule would not be mode correct: we would fail in step 2 because $\Gamma, x{:}\tau_1$ is not known because τ_1 is not known.

For inference rules with multiple premises we analyze the premises from left to right. For each premise we first show that all inputs are known and outputs are free, then assume all outputs are known before checking the next premise. After the last premise has been checked we still have to show that the outputs of the conclusion are all known by now. As an example, consider the rule for function application.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \textit{AppTyp}$$

Applying our technique, checking actually fails:

1. We **assume** that Γ, e_1 and e_2 are known.
2. We **show** that Γ and e_1 are known and τ_2 and τ are free, all which holds.
3. We **assume** that τ_2 and τ are known.
4. We **show** that Γ and e_2 are known and τ_2 is free. This latter check fails, because τ_2 is known at this point.

Consequently have to rewrite the rule slightly. This rewrite should be obvious if you have implemented this rule in ML: we actually first generate a type τ_2' for e_2 and then compare it to the domain type τ_2 of e_1.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2' \quad \tau_2' = \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau} \; \textit{AppTyp}$$

We consider all constitutents of the equality check to be input ($\tau^+ = \sigma^+$). This now checks correctly as follows:

1. We **assume** that Γ, e_1 and e_2 are known.
2. We **show** that Γ and e_1 are known and τ_2 and τ is free, all which holds.
3. We **assume** that τ_2 and τ are known.
4. We **show** that Γ and e_2 are known and τ_2' is free, all which holds.
5. We **assume** that τ_2' is known.
6. We **show** that τ_2 and τ_2' are known, which is true.
7. We **assume** the outputs of the equality to be known, but there are no output so there are no new assumption.
8. We **show** that τ (output in the conclusion) is known, which is true.

Now we can examine other language constructs and typing rules from the same perspective to arrive at a bottom-up inference system for type checking. We forego this exercise here, and instead consider what can be gained by introducing two mutually recursive judgments: one for expressions that have enough information to synthesize a type, and one for situations where we know what type to expect so we propagate it downward in the tree.

$$\begin{array}{ll} \Gamma^+ \vdash e^+ \uparrow \tau^- & e \text{ synthesizes } \tau \\ \Gamma^+ \vdash e^+ \downarrow \tau^+ & e \text{ checks against } \tau \\ \tau^+ \sqsubseteq \sigma^+ & \tau \text{ is a subtype of } \sigma \end{array}$$

The subtype judgment is the same as $\tau \leq \sigma$, except that we omit the rule of transitivity which is not mode correct; the other two look significantly different from a pure synthesis judgment.

Generally, for *constructors* of a type we can propagate the type information *downward* into the term, which means it should be used in the analysis judgment $e^+ \downarrow \tau^+$. Conversely, the *destructors* generate a result of a smaller type from a constituent of larger type and can therefore be used for synthesis, propagating information *upward*.

We consider some examples. First, functions. A function constructor will be checked, and application synthesizes, in accordance with the reasoning above.

$$\frac{\Gamma, x{:}\tau_1 \vdash e \downarrow \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) \downarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \uparrow \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 \downarrow \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) \uparrow \tau_1}$$

Careful checking against the desired modes is required. In particular, the order of the premises in the rule for application is critical so that τ_2 is available to check e_2. Note that unlike in the case of pure synthesis, no subtype checking is required at the application rule. Instead, this must be handled implicitly in the definition of $\Gamma \vdash e_2 \downarrow \tau_2$. In fact, we will need a general rule that mediates between the two directions. This rule replaces subsumption in the general system.

$$\frac{\Gamma \vdash e \uparrow \tau \quad \tau \sqsubseteq \sigma}{\Gamma \vdash e \downarrow \sigma}$$

Note that the modes are correct: Γ, e, and σ are known as inputs in the conclusion. This means that Γ and e are known and τ is free, so the first premise is mode-correct. This yields a τ as output (if successful). This means we can now check if $\tau \sqsubseteq \sigma$, since both τ and σ are known.

For sums, the situation is slightly trickier, but not much. Again, the constructors are checked against a given type.

$$\frac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \mathtt{inl}(e) \downarrow \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \mathtt{inr}(e) \downarrow \tau_1 + \tau_2}$$

For the destructor, we go from $e \uparrow \tau_1 + \tau_2$ to the two assumptions $x_1{:}\tau_1$ and $x_2{:}\tau_2$ in the two branches. These assumptions should be seen as synthesis, variable synthesize their type from the declarations in Γ (which are given).

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x \uparrow \tau}$$

$$\frac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 \downarrow \sigma \quad \Gamma, x{:}\tau_2 \vdash e_2 \downarrow \sigma}{\Gamma \vdash \mathsf{case}(e, x_1.e_1, x_2.e_2) \downarrow \sigma}$$

Here, both branches are checked against the same type σ. This avoids the need for computing the least upper bound, because one branch might synthesize σ_1, the other σ_2, but they are checked separately against σ. So σ must be an upper bound, but since we don't have to synthesize a principal type we never need to compute the least upper bound.

Finally, we consider recursive types. The simple idea that constructors (here: `roll`) should be checked against a type and destructors (here: `unroll`) should synthesize a type avoids any annotation on the type.

$$\frac{\Gamma \vdash e \downarrow \{\mu t.\sigma/t\}\sigma}{\Gamma \vdash \mathsf{roll}(e) \downarrow \mu t.\sigma} \qquad \frac{\Gamma \vdash e \uparrow \mu t.\sigma}{\Gamma \vdash \mathsf{unroll}(e) \uparrow \{\mu t.\sigma/t\}\sigma}$$

This seems too good to be true, because so far we have not needed *any* type information in the terms! However, there are still a multitude of situations where we need a type, namely where an expression requires a type to be checked, but we are in synthesis mode. Because of our general philosophy, this happens precisely where a destructor is meets a constructors, that is, where we can apply reduction in the operational semantics! For example, in the expression

```
(fn x => x) 3
```

the function part of the application is required to synthesize, but `fn x => x` can only be checked.

The general solution is to allow a type annotation at the place where synthesis and analysis judgments meet in the opposite direction from the subsumption rule shown before. This means we require a new form of syntax, $e : \tau$, and this is the *only* place in an expression where a type needs to occur. Then the example above becomes

```
(fn x => x : int -> int) 3
```

From this example it should be clear that bidirectional checking is not necessarily advantageous over pure synthesis, at least with the simple strategy we have employed so far.

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau}$$

Looking back at our earlier example, we obtain:

$$\begin{array}{lcl} \mathsf{nat} & = & \mu t.1\texttt{+}t \\ \mathsf{zero} & = & \texttt{roll}(\texttt{inl}(\texttt{unitel})) : \mathsf{nat} \\ \mathsf{succ} & = & \texttt{fn}(x.\texttt{roll}(\texttt{inr}(x))) : \mathsf{nat} \to \mathsf{nat} \end{array}$$

One reason this seems to work reasonably well in practice that code rarely contains explicit redexes. Programmers instead tend to turn them into definitions, which then need to be annotated. So the rule of thumb is that in typical programs one needs to annotate the outermost functions and recursions, and the local functions and recursions, but not much else.

With these ideas in place, one can prove a general soundness and completeness theorem with respect to the original subtyping system. We will not do this here, but move on to discuss the form of subtyping that is amenable to an algorithmic interpretation. In other words, we want to write out a judgment $\tau \sqsubseteq \sigma$ which holds if and only if $\tau \leq \sigma$, but which is mode-correct when both τ and σ are given.

The difficulty in the ordinary subtyping rules is transitivity

$$\frac{\tau \leq \sigma \quad \sigma \leq \rho}{\tau \leq \rho}\ \textit{Trans}$$

which is not well-moded: σ is an input in the premise, but unknown. So we have to design a set of rules that get by without the rule of transitivity. We write this new judgment as $\tau \sqsubseteq \sigma$. The idea is to eliminate transitivity and reflexivity and just have decomposition rules except for the primitive coercion from int to float.[1] We will not write the coercions explicitly for the

[1] In Assignment 6, a slightly different choice has been made to account for type variables which we ignore here.

sake of brevity.

$$\frac{}{\mathsf{int} \sqsubseteq \mathsf{float}}$$

$$\frac{}{\mathsf{int} \sqsubseteq \mathsf{int}} \qquad \frac{}{\mathsf{float} \sqsubseteq \mathsf{float}} \qquad \frac{}{\mathsf{bool} \sqsubseteq \mathsf{bool}}$$

$$\frac{\sigma_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \to \tau_2 \sqsubseteq \sigma_1 \to \sigma_2}$$

$$\frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \times \tau_2 \sqsubseteq \sigma_1 \times \sigma_2} \qquad \frac{}{1 \sqsubseteq 1}$$

$$\frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 + \tau_2 \sqsubseteq \sigma_1 + \sigma_2} \qquad \frac{}{0 \sqsubseteq 0}$$

Note that these are well-moded with $\tau^+ \sqsubseteq \sigma^+$. We have ignored here universal, existential and recursive types: adding them requires some potentially difficult choices that we would like to avoid for now.

Now we need to show that the algorithmic formulation of subtyping ($\tau \sqsubseteq \sigma$) coincides with the original specification of subtyping ($\tau \leq \sigma$). We do this in several steps.

Lemma 1 (Soundness of algorithmic subtyping)
If $\tau \sqsubseteq \sigma$ then $\tau \leq \sigma$.

Proof: By straightforward induction on the structure of the given derivation. ■

Next we need two properties of algorithmic subtyping. Note that these arise from the attempt to prove the completeness of algorithmic subtyping, but must nonetheless be presented first.

Lemma 2 (Reflexivity and transitivity of algorithmic subtyping)
(i) $\tau \sqsubseteq \tau$ for any τ.

(ii) If $\tau \sqsubseteq \sigma$ and $\sigma \sqsubseteq \rho$ then $\tau \sqsubseteq \rho$.

Proof: For (i), by induction on the structure of τ.

For (ii), by simultaneous induction on the structure of the two given derivations $\mathcal{D}$ of $\tau \sqsubseteq \sigma$ and $\mathcal{E}$ of $\sigma \sqsubseteq \rho$. We show one representative cases; all others are similar or simpler.

Case: $\mathcal{D} = \dfrac{\sigma_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \to \tau_2 \sqsubseteq \sigma_1 \to \sigma_2}$ and $\mathcal{E} = \dfrac{\rho_1 \sqsubseteq \sigma_1 \quad \sigma_2 \sqsubseteq \rho_2}{\sigma_1 \to \sigma_2 \sqsubseteq \rho_1 \to \rho_2}$. Then

$\rho_1 \sqsubseteq \tau_1$	By i.h.
$\tau_2 \sqsubseteq \rho_2$	By i.h.
$\tau_1 \to \tau_2 \sqsubseteq \rho_1 \to \rho_2$	By rule

■

Now we are ready to prove the completeness of algorithmic subtyping.

Lemma 3 (Completeness of algorithmic subtyping)
If $\tau \leq \sigma$ then $\tau \sqsubseteq \sigma$.

Proof: By straightforward induction over the derivation of $\tau \leq \sigma$. For reflexivity, we apply Lemma 2, part (i). For transitivity we appeal to the induction hypothesis and apply Lemma 2, part (ii). In all other cases we just apply the induction hypothesis and then the corresponding algorithmic subtyping rule. ■

Summarizing the results above we obtain:

Theorem 4 (Correctness of algorithmic subtyping)
$\tau \leq \sigma$ if and only if $\tau \sqsubseteq \sigma$.

Lecture Notes on Mutable Storage

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 16
October 26, 2004

After several lectures on extensions to the type system that are independent from computational mechanism, we now consider mutable storage as a computational effect. This is a counterpart to the study of exceptions and continuations which are *control effects*.

We will look at mutable storage from two different points of view: one, where essentially all of MinML becomes an imperative language and one where we use the type system to isolate effects (next lecture). The former approach is taken in ML, that latter in Haskell.

To add effects in the style of ML, we add a new type τ ref and three new expressions to create a mutable cell ($\mathsf{ref}(e)$), to write to the cell ($e_1 := e_2$), and read the contents of the cell ($!e$). There is only a small deviation from the semantics of Standard ML here in that updating a cell returns its new value instead of the unit element. We also need to introduce cell labels themselves so we can uniquely identify them. We write l for *locations*. Locations are assigned types in a *store typing* Λ.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Locations never appear in the input program, but they can arise during evaluation, when cells are allocated using the $\mathsf{ref}(e)$ construct. We therefore need to thread the store typing through the typing judgment which now has the form $\Lambda; \Gamma \vdash e : \tau$. We obtain the following rules, which should be familiar from Standard ML. We use here the concrete, rather than the abstract syntax, in order to present the assignment and dereferencing operations.

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \mathsf{ref}(e) : \tau\ \mathsf{ref}} \qquad \frac{\Lambda;\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Lambda;\Gamma \vdash e_2 : \tau}{\Lambda;\Gamma \vdash e_1 := e_2 : \tau}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\ \mathsf{ref}}{\Lambda;\Gamma \vdash !e : \tau} \qquad \frac{l{:}\tau \text{ in } \Lambda}{\Lambda;\Gamma \vdash \mathsf{loc}(l) : \tau\ \mathsf{ref}}$$

It is important to keep in mind the difference between locations and variables. Expressions that we evaluate are always closed with respect to variables (we substitute for them), but they may contain references l to locations.

To describe the operational semantics, we need to model the store. We think of it simply as a mapping from locations to values and we denote it by M for memory.

$$\textit{Stores} \qquad M ::= \cdot \mid M, l{=}v$$

Note that in the evaluation of a functional program in a real compiler there are many other uses of memory (heap and stack, for example), while the store only contains the mutable cells. As usual, we assume that all locations in a store are distinct.

In this approach to modeling mutable storage, the evaluation of any expression can potentially have an effect. This means we need to change our basic model of computation to add a store. We replace the ordinary transition judgment $e \mapsto e'$ by

$$\langle M, e\rangle \mapsto \langle M', e'\rangle$$

which asserts that expression e in store M steps to expression e' with store M'. First, we have to take care of changing *all* prior rules to thread through the store. Fortunately, this is quite systematic. We show only the cases for functions.

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, \mathtt{apply}(e_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(e_1', e_2)\rangle}$$

$$\frac{v_1\ \mathsf{value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, \mathtt{apply}(v_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(v_1, e_2')\rangle}$$

$$\frac{v_2\ \mathsf{value}}{\langle M, \mathtt{apply}(\mathtt{fn}(\tau_2, x.e), v_2)\rangle \mapsto \langle M, \{v_2/x\}e\rangle}$$

For the new operations we have to be careful about the evaluation order, and also take into account that evaluating, say, the initializer of a new cell may actually change the store.

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M', \mathsf{ref}(e')\rangle} \qquad \frac{v \ \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), \mathsf{loc}(l)\rangle} \qquad \frac{}{\mathsf{loc}(l) \ \ \mathsf{value}}$$

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, e_1 := e_2\rangle \mapsto \langle M', e_1' := e_2\rangle} \qquad \frac{v_1 \ \ \mathsf{value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, v_1 := e_2\rangle \mapsto \langle M', v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2)}{\langle M, \mathsf{loc}(l) := v_2\rangle \mapsto \langle M', v_2\rangle}$$

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, !e\rangle \mapsto \langle M', !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !\mathsf{loc}(l)\rangle \mapsto \langle M, v\rangle}$$

In order to state type preservation and progress we need to define well-formed machine states which in turn requires validity for the memory configuration. For that, we need to check that each cell contains a value of the type prescribed by the store typing. The value stored in each cell can reference other cells which can in turn refer back to the first cell. In other words, the pointer structure of memory can be cyclic. We therefore need to check the contents of each cell knowing the typing of all locations. The judgment has the form $\Lambda_0; \cdot \vdash M : \Lambda$, where we intend Λ_0 to range over the whole store typing will we verify on the right-hand side that each cell has the prescribed type.

$$\frac{}{\Lambda_0; \cdot \vdash (\cdot) : (\cdot)} \qquad \frac{\Lambda_0; \cdot \vdash M : \Lambda \quad \Lambda_0; \cdot \vdash v : \tau \quad v \ \ \mathsf{value}}{\Lambda_0; \cdot \vdash (M, l{=}v) : (\Lambda, l{:}\tau)}$$

With this defined, we can state appropriate forms of type preservation and progress theorems. We write $\Lambda' \geq \Lambda$ if Λ' is an extension of the store typing Λ with some additional locations. In this particular case, for a single step, we need at most one new location so that if $\Lambda' \geq \Lambda$ then either $\Lambda' = \Lambda$ or $\Lambda' = \Lambda, l{:}\tau$ for a new l and some τ.

Theorem 1 (Type Preservation)
If $\Lambda; \cdot \vdash e : \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *and* $\langle M, e\rangle \mapsto \langle M', e'\rangle$ *then for some* $\Lambda' \geq \Lambda$ *and memory* M' *we have* $\Lambda'; \cdot \vdash e' : \tau$ *and* $\Lambda'; \cdot \vdash M' : \Lambda'$.

Proof: By induction on the derivation of the computation judgment, applying inversion on the typing assumptions. ■

Theorem 2 (Progress)
If $\Lambda; \cdot \vdash e : \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *then either*

(i) e value*, or*

(ii) $\langle M, e\rangle \mapsto \langle M', e'\rangle$ *for some* M' *and* e'*.*

Proof: By induction on the derivation of the typing judgment, analyzing all possible cases. ■

We assume the reader is already familiar with the usual programming idioms using references and assignment. As an example that illustrates one of the difficulties of reasoning about programs with possibly hidden effect, consider the following ML code.

```
signature COUNTER =
sig
  type c
  val new : int -> c  (* create a counter *)
  val inc : c -> int  (* inc and return new value *)
end;
structure C :> COUNTER =
struct
  type c = int ref
  fun new(n):c = ref(n)
  fun inc(r) = (r := !r+1; !r)
end;
val c = C.new(0);
val 1 = C.inc(c);
val 2 = C.inc(c);
```

Here the two calls to `C.inc(c)` are identical but yield different results. This is the intended behavior, but clearly not exposed in the type of the expressions involved. There are many pitfalls in programming with ephemeral data structures that most programmers are all too familiar with.

The way we have extended MinML with mutable storage has several drawbacks. The principal difficulty with programming with effects is that the type system does not track them properly. So when we examine the type of a function $\tau_1 \to \tau_2$ we cannot tell if the function simply returns a value of type τ_2 or if it could also have an effect. This complicates reasoning about programs and their correctness tremendously.

An alternative is to try to express in the type system that certain functions may have effects, while others do not have effects. This is the purpose of *monads* that are quite popular in the Haskell community. Haskell is a lazy[1] functional language in which all effects are isolated in a monad. We will see that monadic programming has its own drawbacks. The last word in the debate on how to integrate imperative and pure functional programming has not yet been spoken.

We introduce monads in two steps. The first step is the generic framework, which can be instantiated to different kinds of effects. In this lecture we introduce mutable storage as an effect, just as we did in the previous lecture on mutable storage in ML.

In the generic framework, we extend MinML by adding a new syntactic category of *monadic expressions*, denoted by m.[2] Correspondingly, there is a new typing judgment

$$\Gamma \vdash m \div \tau$$

expressing that the monadic expression m has type τ in context Γ. We think of a monadic expression as one whose evaluation returns not only a value of type τ, but also may have an effect. We introduce this separate category so that the ordinary expressions we have used so far can remain pure, that is, free of effects.

Any particular use of the monadic framework will add particular new monadic expressions, and also possibly new pure expressions. But first the constructs that are independent of the kind of effect we want to consider. The first principle is that a pure expression e can be considered as a monadic expression e which happens to have no effect.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau}$$

The second idea is that we can quote a monadic expression and thereby turn it into a pure expression. It has no effects because the monadic expression will not be executed. We write the quotation operator as $\mathsf{comp}(m)$, and the type of quoted computations of type τ as $\bigcirc\tau$.

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{comp}(m) : \bigcirc\tau} \qquad \frac{}{\mathsf{comp}(m)\ \ \mathsf{value}}$$

[1] Lazy here means call-by-name with memoization of the suspension.

[2] In lecture, we did not use a separate syntactic category, but just as writing v for values aids understanding, writing m for potentially effectful expressions makes it easier to interpret some rules.

Finally, we must be able to unwrap and thereby actually execute a quoted monadic expression. However, we cannot do this anywhere in a pure expression, because evaluating such a supposedly pure expression would then have an effect. Instead, we can only do this if we are within an explicit sequence of monadic expressions! This yields the following construct

$$\frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{letcomp}(e, x.\, m) \div \sigma}$$

We will use the concrete syntax $\mathsf{let\ comp}\, x = e\, \mathsf{in}\, m\, \mathsf{end}$ for $\mathsf{letcomp}(e, x.\, m)$. Note that m and $\mathsf{letcomp}(e, x.\, m)$ are monadic expressions and therefore may have an effect, while e is a pure expression of monadic type. We think of the effects are being staged as follows:

(1) We evaluate e which should yield a value $\mathsf{comp}(m')$.

(2) We execute the monadic expression m', which will have some effects but also return a value v.

(3) Substitute v for x in m and then execute the resulting monadic expression.

In order to specify this properly we need to be able to describe the effect that may be engendered by executing a monadic expression. The judgment for executing monadic expressions then has the form

$$\langle M, m\rangle \mapsto \langle M', m'\rangle$$

where the store changes from M to M' and the expression steps from m to m'. According to the considerations above, we obtain first the following rules, where we use a pure expression as a (trivial) form of monadic expression.

$$\frac{e\ \mathsf{pure} \quad e \mapsto e'}{\langle M, e\rangle \mapsto \langle M, e'\rangle}$$

Here we have used $e\ \mathsf{pure}$ for the judgment that e can be classified by the typing rules as $e : \tau$. Just like the property of being a value, this is a purely syntactic property of e. Furthermore, it is shallow: letcomp, allocation, assignment, and dereference are monadic expressions while all others are pure.

We can see that the transition judgment on ordinary expressions looks the same as before and that it can have no effect. Contrast this with the

situation in ML from the previous lecture where we needed to change *every* transition rule to account for possible effects.

The next sequence of three rules implement items (1), (2), and (3) above.

$$\frac{e \mapsto e'}{\langle M, \mathsf{letcomp}(e, x.\, m)\rangle \mapsto \langle M, \mathsf{letcomp}(e', x.\, m)\rangle}$$

$$\frac{\langle M, m_1\rangle \mapsto \langle M', m_1'\rangle}{\langle M, \mathsf{letcomp}(\mathsf{comp}(m_1), x.\, m)\rangle \mapsto \langle M', \mathsf{letcomp}(\mathsf{comp}(m_1'), x.\, m)\rangle}$$

$$\frac{v \;\; \mathsf{value}}{\langle M, \mathsf{letcomp}(\mathsf{comp}(v), x.\, m)\rangle \mapsto \langle M, \{v/x\}m\rangle}$$

Note that the substitution in the last rule is appropriate. The substitution principle for pure values into monadic expressions is straightforward precisely because v is cannot have effects.

We will not state here the generic forms of the preservation and progress theorems. They are somewhat trivialized because our language, while designed with effects in mind, does not yet have any actual effects.

In order to define the monad for mutable storage we introduce a new form of type, $\tau\,\mathsf{ref}$ and three new forms of monadic expressions, namely $\mathsf{ref}(e)$, $e_1 := e_2$ and $!e$. In addition we need one new form of pure expression, namely locations l which are declared in a store typing Λ with their type. Recall the form of store typings.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Locations can be pure because creating, assigning, or dereferencing them is an effect, and the types prevent any other operations on them. The store typing must now be taking into account when checking expressions that are created a runtime. They are, however, not needed for compile-time checking because the program itself, before it is started, cannot directly refer to locations. We just uniformly add "Λ;" to all the typing judgments—they are simply additional hypotheses of a slightly different form than what is recorded in Γ.

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \mathsf{ref}(e) \div \tau\,\mathsf{ref}} \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau\,\mathsf{ref} \quad \Lambda; \Gamma \vdash e_2 : \tau}{\Lambda; \Gamma \vdash e_1 := e_2 \div \tau}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau\,\mathsf{ref}}{\Lambda; \Gamma \vdash !e \div \tau} \qquad \frac{l{:}\tau \text{ in } \Lambda}{\Lambda; \Gamma \vdash \mathsf{loc}(l) : \tau\,\mathsf{ref}}$$

Note that the constituents of the new monadic expressions are pure expressions. This guarantees that they cannot have effects: all effects must be explicitly sequenced using the letcomp form.

Now the additional rules for new expressions are analogous to those we had when effects where not encapsulated in the monad.

$$\frac{e \mapsto e'}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M, \mathsf{ref}(e')\rangle} \qquad \frac{v \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), \mathsf{loc}(l)\rangle} \qquad \frac{}{l \ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\langle M, e_1 := e_2\rangle \mapsto \langle M, e_1' := e_2\rangle} \qquad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\langle M, v_1 := e_2\rangle \mapsto \langle M, v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2) \quad v_2 \ \mathsf{value}}{\langle M, \mathsf{loc}(l) := v_2\rangle \mapsto \langle M', v_2\rangle}$$

$$\frac{e \mapsto e'}{\langle M, !e\rangle \mapsto \langle M, !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !\mathsf{loc}(l)\rangle \mapsto \langle M, v\rangle}$$

We complete this lecture with a simple example. In the next lecture we discuss the progress and preservation properties and other forms of effects. In MinML with pervasive effects, we might write the following, which allocates a cell initialized with 3 and then increments it.

```
let x = ref 3
in x := !x + 1 end;
```

In MinML with effects encapsulated in a monad, we would rewrite this as follows.

```
let comp x = comp (ref 3) in
let comp y = comp (!x) in
let comp z = comp (x := y+1) in
z end end end
```

Note that the arguments to assignment must be pure expressions, so we must explicitly sequence the computation into two assignments.

It is this rewriting of expressions which is often required that can make programming with effects in monadic style tedious (although some syntactic sugar can clearly help). Another problem is that operations such as input/output are also effects and therefore must be inside the monad.

This means that inserting a print statement into a function changes its type, which can complicate debugging.

Lecture Notes on Monadic Input and Output

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 17
October 28, 2004

After reviewing the basic idea behind the encapsulation of effects, we introduce input and output as a specific kind of effect. After store effects in the last lecture, this will be our second example. For simplicity, we don't consider store and input/output simultaneously.

In review, in a pure functional language programs are evaluated only to obtain a value. A loose characterization of an effect is simply everything else that a function might perform. Allocating, mutating, and reading storage cells is one example of effects, input and output are two more. We say that effects are *encapsulated* if they do not interfere with the meaning of the pure expressions in a language. Standard ML does not encapsulate effects, Haskell does.

Encapsulation of effects is achieved by separating *pure expressions* (e) from potentially *effectful expressions* (written as m). All the usual constructs in MinML remain pure expressions.

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \tau_1 \rightarrow \tau_2 \mid \cdots \mid \tau \ \mathsf{ref} \mid \bigcirc\tau \\ \text{Pure Expressions} & e & ::= & \mathsf{fn}(x.e) \mid \mathsf{apply}(e_1, e_2) \mid \cdots \mid \mathsf{comp}(m) \\ \text{Monadic Expressions} & m & ::= & e \mid \mathsf{letcomp}(e, x.m) \\ & & & \mid \mathsf{ref}(e) \mid \mathsf{assign}(e_1, e_2) \mid \mathsf{deref}(e) \end{array}$$

In the concrete syntax, we write τ `comp` for $\bigcirc\tau$ and let comp $x = e$ in m end for $\mathsf{letcomp}(e, x.\, m)$. As an example, consider the following signature and implementation.

```
signature C =
sig
```

```
  type c
  val new : c comp
  val inc : c -> unit comp
  val get : c -> int comp
end;

structure C :> C =
struct
  type c = int ref
  val new = comp (ref 0)
  val inc = fn r => comp (let comp x = !r in r := x+1 end)
  val get = fn r => comp (!r)
end;
```

Now we can create a cell, increment and read it with the following (monadic) expression, using a slight shorthand by allowing multiple declarations as in Standard ML.

```
let
  comp x = C.new
  comp _ = C.inc x
  comp y = C.get x
in
  y
end;
```

When started in the empty memory, the above monadic expression executes and evaluates to $\langle (l{=}1), 1\rangle$ for some l. It is worth writing out this computation step by step to see exactly how computation proceeds and effects and effect-free computations may be interleaved.

In order to model input and output, the state that monadic expressions may refer to contains two streams: an input stream and an output stream. Streams are potentially infinite sequence of integers, $k_1 \cdots k_n \cdots$. The empty stream is denoted by ϵ. We denote streams by s and write (s_I, s_O) for the pair of input and output streams. We have the constructs read, eof and $\mathsf{write}(e)$ for reading from the input stream, testing if the input stream is empty, and writing the value of e to the output stream, respectively. These constructs must be monadic expressions, since they have an effect.

$$\frac{}{\Gamma \vdash \mathsf{read} \div \mathsf{int}}$$

$$\frac{}{\Gamma \vdash \mathsf{eof} \div \mathsf{bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{write}(e) \div 1}$$

As in the case of mutable storage, the operational semantics distinguishes states for monadic expressions (which must include the input and output streams) and pure expressions (which does not include the input and output streams).

$$\frac{}{\langle (k \cdot s_I, s_O), \mathsf{read} \rangle \mapsto \langle (s_I, s_0), \mathsf{int}(k) \rangle} \; \textit{Read}$$

$$\frac{}{\langle (k \cdot s_I, s_O), \mathsf{eof} \rangle \mapsto \langle (k \cdot s_I, s_O), \mathsf{false} \rangle} \; \textit{EofFalse}$$

$$\frac{}{\langle (\epsilon, s_O), \mathsf{eof} \rangle \mapsto \langle (\epsilon, s_O), \mathsf{true} \rangle} \; \textit{EofTrue}$$

$$\frac{e \mapsto e'}{\langle (s_I, s_O), \mathsf{write}(e) \rangle \mapsto \langle (s_I, s_O), \mathsf{write}(e') \rangle} \; \textit{WriteArg}$$

$$\frac{v \;\; \mathsf{value}}{\langle (s_I, s_O), \mathsf{write}(\mathsf{int}(k)) \rangle \mapsto \langle (s_I, k \cdot s_O), \langle\,\rangle \rangle} \; \textit{Write}$$

This language extension seems simple, although it is not completely trivial to write programs using the monadic syntax. Moreover, we have to formulate the progress theorem carefully. In fact, with the stated rules it fails! The reason is that if we try to read from an empty input stream no rule is applicable. It is quite instructive to see where the proof of progress fails unless we incorporate the possibility that m may be blocked. In a concurrent language (or even in a realistic sequential language) such blocking on input can certainly occur, so it seems reasonable to allow for it and model it. Of course, our language has no explicit mechanism of unblocking, but this will change later on. So we are aiming at the following version of the progress theorem.

Theorem 1 (Progress)

(1) If $\cdot \vdash e : \tau$ *then either*

(i) $e \mapsto e'$ *for some* e'*, or*

(ii) e value

(2) If $\cdot \vdash m \div \tau$ *then either*

(i) $\langle (s_I, s_O), m\rangle \mapsto \langle (s_I', s_O'), m'\rangle$ *for some* s_I'*,* s_O' *and* m'*, or*

(ii) $m = v$ *and* v value*, or*

(iii) $s_I = \epsilon$ *and* m blocked*.*

The first thought on how to define the judgment m blocked would be to simply write

$$\frac{}{\mathsf{read\ blocked}}\ \textit{BlockedRead}$$

However, this is not enough as, again, a failed proof of the progress theorem should tell you. We may also be in the situation where the read is not at the top level, but is the first monadic expression to be executed. In other words, the search rules may lead us to a read expression. The general way to capture this is with the rule

$$\frac{m_1\ \mathsf{blocked}}{\mathsf{letcomp}(\mathsf{comp}(m_1), x.m_2)\ \mathsf{blocked}}\ \textit{BlockedLet}$$

Note that we never need to look at m_2, nor do we need to account for the case of $\mathsf{letcomp}(e, x.m_2)$ where e is not a value, because a pure expression e cannot have an effect unless it is situated as in the *BlockedLet* rule.

With the right definition of blocked states, it is then easy to prove the progress theorem, employing value inversion as in other progress proofs we have carried out up to this point. We just have to be sure to cover all the possible cases.

We close the lecture with two simple examples. The first is a (non-recursive) computation which reads one integer and then writes to the output.

```
val copyOne : unit comp =
  comp (let comp x = read in write x end)
```

The next one is a recursive function to copy the whole input stream to the output stream. This function should loop forever, if the input stream is infinite.

```
val copy : unit comp =
  rec copy =>
    comp (let
            comp b = eof
            comp x = if b then comp (let
                                       comp _ = copyOne
                                       comp _ = copy
                                     in () end)
                          else comp ()
          in () end)
```

This last example has some subtleties. For example, the conditional is a pure expression (and not a monadic expression). In order to properly interleave pure and effectful computation, it must be essentially where it is: on the right-hand side of a `comp` declaration, where both branches are again computations.

Lecture:
Records and Variants

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 2, 2004

1 Records

One disadvantage of using tuples to aggregate together many pieces of data is that it requires the programmer to use a possibly long — and certainly opaque — sequence of `fst`s and `snd`s to extract the desired component. Even in a realistic language with features like pattern-matching, the programmer using a tuple would need to at least remember in which order all of the elements come. It may be especially hard to remember the meaning of the individual elements if many are of the same type.

A solution to this problem is to introduce *labelled records*. We assume that there is an infinite supply of *labels* ℓ (which we may imagine as, for instance, strings) and extend our language as follows:

$$\begin{array}{rrcll} \textit{Types} & \tau & ::= & \cdots & \mid \{\bar{\ell} : \bar{\tau}\} \\ \textit{Values} & v & ::= & \cdots & \mid \{\bar{\ell} = \bar{v}\} \\ \textit{Expressions} & e & ::= & \cdots & \mid \{\bar{\ell} = \bar{e}\} \\ & & & & \mid \#\ell(e) \end{array}$$

Where $\bar{\ell} : \bar{\tau}$ is shorthand for $\ell_1 : \tau_1, \ldots \ell_n : \tau_n$ and similarly for $\bar{\ell} = \bar{e}$ and $\bar{\ell} = \bar{v}$.

The expression $\{\bar{\ell} = \bar{e}\}$ constructs a record where, for each i the ℓ_i field of the record is assigned the expression e_i. The deconstructor is the record projection $\#\ell(e)$, which is like the tuple projections `fst` and `snd`, except that it requests the named field ℓ from the record expression e. A record expression is a value just in case all of its fields are assigned expressions which happen to be values.

A record is well-typed if all of its components are, and a labelled projection is well-typed if its argument is a record that contains that label. Formally, the typing rules for the new constructs are as follows.

$$\frac{\Gamma \vdash e_i : \tau_i \qquad \text{(for all } i\text{)}}{\Gamma \vdash \{\bar{\ell} = \bar{e}\} : \{\bar{\ell} : \bar{\tau}\}} \qquad \frac{\Gamma \vdash e : \{\bar{\ell} : \bar{\tau}\}}{\Gamma \vdash \#\ell_i(e) : \tau_i}$$

To evaluate a record, we evaluate all of its components in the order they are given. To evaluate a projection, we evaluate the expression it projects. When a projection applies to a record value, we return the value with the label of the projection. We give the formal evaluation rules in small-step style.

$$\frac{e_i \mapsto e_i'}{\{\bar{\ell} = \bar{v}, \ell_i = e_i, \bar{\ell}'' = \bar{e}''\} \mapsto \{\bar{\ell} = \bar{v}, \ell_i = e_i', \bar{\ell}'' = \bar{e}''\}} \qquad \frac{e \mapsto e'}{\#\ell(e) \mapsto \#\ell(e')}$$

$$\frac{}{\#\ell_i(\{\bar{\ell} = \bar{v}\}) \mapsto v_i}$$

It is a natural question to ask how records can be subtyped. In fact, though they are in a sense 'merely' a generalization of tuples, they admit richer subtyping properties. The subtyping that we expect to be able to do from our experience with tuples is still present, and we refer to it as *depth subtyping* of records. The idea is that if we have two record types with the same labels, but of different types, if the types of one record are individually all subtypes of the other's, then the one record type as a whole is a subtype of the other.

$$\frac{\tau_i \leq \tau_i' \qquad \text{(for all } i\text{)}}{\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell} : \bar{\tau}'\}}$$

If we fix a particular record type $\{\bar{\ell} : \bar{\tau}\}$ and a list of coercions $f_1 : \tau_1 \leq \tau_1', \ldots, f_n : \tau_n \leq \tau_n'$, then the coercion that witnesses $\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell} : \bar{\tau}'\}$ can be written as

```
fn record : {lab1 : tau1, ..., labn : taun} =>
 {lab1 : f1(#lab1(record)),
  lab2 : f2(#lab2(record)),
  ...
  labn : fn(#labn(record))}
```

However, remember that $\tau \leq \tau'$ in general can be interpreted as *an expression of type τ can safely be substituted in a place that expects an expression of type τ'*. Since all the information we get out of records is by projecting their fields, it cannot hurt us if the record we are handed has 'too many' fields. As long as it has the fields that we want, more fields besides those are acceptable. Thus a record type that includes more fields than another, without changing any types of the original, is a subtype of it. This is called *width subtyping*, since a record with more fields is seen as 'wider.'

$$\overline{\{\bar{\ell} : \bar{\tau}, \bar{\ell}' : \bar{\tau}'\} \leq \{\bar{\ell} : \bar{\tau}\}}$$

The coercion function for this subtyping can be written as

```
fn record :
  {lab1  : tau1,  ..., labn  : taun,
   lab1' : tau1', ..., labm' : taum'} =>
 {lab1 : #lab1(record),
  lab2 : #lab2(record),
  ...
  labn : #labn(record)}
```

Finally, we would like to be able to say that two record types are isomorphic if they only differ in the order of their fields.

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell'_i \qquad \tau_{\pi i} = \tau'_i \qquad (\text{for all } i)}{\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell}' : \bar{\tau}'\}}$$

Since if π is a permutation, so too is its inverse π^{-1}, and we have that if one record type is a permuation of the other, then both are subtypes of each other, i.e. isomorphic.

The coercion function (assuming `lab1'`... `labn'` are a permutation of the original labels `lab1`... `labn`) is

```
fn record : {lab1 : tau1,  ..., labn : taun} =>
 {lab1' : #lab1'(record),
  lab2' : #lab2'(record),
  ...
  labn' : #labn'(record)}
```

However, if we look back to the development of the idea of subtyping, we recall that it is important that we don't need to add a separate rule for transitivity. This is because having a transitivity rule would make it unclear how to write an algorithm for checking whether subtyping holds, since we don't know how to guess the 'middle' type that appears in both premises but not in the conclusion.

The rules we have given so far for record subtyping do not admit transitivity as an admissible rule. That is, they themselves don't allow us to derive as many facts as we would be able to if we also had a transitivity rule. For instance, if we had transitivity, we could conclude that $\{x : int, y : int, z : int\} \leq \{y : float, x : int\}$ by using depth subtyping to change y's type from int to float, width subtyping to drop the field z, and permutation subtyping to interchange the positions of x and y. We cannot actually show this with just the three rules above.

To fix this, we introduce a single record subtyping rule that encompasses all three forms of record subtyping at once:

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell''_i \qquad \tau_{\pi i} \leq \tau''_i \qquad (\text{for all } i)}{\{\bar{\ell} : \bar{\tau}, \bar{\ell'} : \bar{\tau'}\} \leq \{\bar{\ell''} : \bar{\tau''}\}}$$

If we fix a particular record type $\{\bar{\ell} : \bar{\tau}\}$ and a list of coercions $f_1 : \tau_{\pi 1} \leq \tau''_1, \ldots, f_n : \tau_{\pi n} \leq \tau''_n$, and assume that `lab1'`...`labn'` are a permutation of the original labels `lab1`...`labn` via π, then then the coercion that witnesses the resulting fact can be written as

```
fn record : {lab1  : tau1,  ..., labn  : taun,
             lab1' : tau1', ..., labn' : taum'} =>
 {lab1' : f1(#lab1'(record)),
  lab2' : f2(#lab2'(record)),
  ...
  labn' : fn(#labn'(record))}
```

2 Variants

Just as binary sum types express the alternation between two possibilities in a way dual to the combination of two pieces of data of binary product types, we can take records — which combine an arbitrary number of

smaller pieces of data — and describe their dual, the concept of *named variants* which allows for the alternation among arbitrary number of possibilities. Named variants are familiar, if not under that name, to any ML programmer. In ML, a `datatype` declaration that doesn't use polymorphism or recursive types is just a declaration of a named variant. We extend our language as follows:

$$\begin{array}{rrcll} \textit{Types} & \tau & ::= & \cdots & \mid \langle \bar{\ell} : \bar{\tau} \rangle \\ \textit{Values} & v & ::= & \cdots & \mid \langle \ell = v \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \\ \textit{Expressions} & e & ::= & \cdots & \mid \langle \ell = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \\ & & & & \mid \mathtt{case}(e, \bar{x}.\bar{e}) \end{array}$$

where again we use shorthand like $\bar{\ell} : \bar{\tau}$ for $\ell_1 : \tau_1, \ldots \ell_n : \tau_n$ and $\bar{x}.\bar{e}$ for $x_1.e_1, x_2.e_2, \ldots, x_n.e_n$. Like records, variant types are specified by giving a list of labels and types. Here the list means a set of possibilities rather than a set of fields. Consequently, the expressions and values of a variant type only contain one label and one expression. In order for these expressions to have unique types, we add type annotations, just as we did for sum types. Also familiar from sum types is the destructor, a case construct. The difference is that the case has arbitrarily many branches, instead of just two.

A variant expression is well-typed if it contains an expression which is well-typed at the branch of the variant type corresponding to the given label. A case statement is well-typed if the expression being cased over is of variant type, and all the branches, when given a variable of the type of one of the branches of the variant, share the same result type. Formally, the typing rules are:

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \langle \ell_i = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} : \langle \bar{\ell} : \bar{\tau} \rangle} \qquad \frac{\Gamma \vdash e : \langle \bar{\ell} : \bar{\tau} \rangle \qquad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad \text{(for all } i\text{)}}{\Gamma \vdash \mathtt{case}(e, \bar{x} : \bar{e}) : \sigma}$$

The evaluation rules work very much like those for sum types. In both the constructor and the first argument of the deconstructor, as usual, we evaluate expressions until they become values. A case statement applied to a variant value v then takes the appropriate branch, according to the label of v, and the expression of v is substituted for the branch's variable.

$$\frac{e \mapsto e'}{\langle \ell_i = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \mapsto \langle \ell_i = e' \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle}} \qquad \frac{e \mapsto e'}{\mathtt{case}(e, \bar{x}.\bar{e}) \mapsto \mathtt{case}(e', \bar{x}.\bar{e})}$$

$$\frac{}{\mathtt{case}(\langle \ell_i = v \rangle, \bar{x}.\bar{e}) \mapsto \{v/x\}e_i}$$

A noticeable difference between variants and records is that variants, as presented above, require a type annotation on every variant expression. However, if we introduce subtyping (especially if we are working in a system of bidirectional typechecking) we can relax the notion that every expression must have a unique type, and drop the type annotation. In this case the syntax for expressions and values is simply

$$\begin{array}{rrcll} \textit{Values} & v & ::= & \cdots & \mid \langle \ell = v \rangle \\ \textit{Expressions} & e & ::= & \cdots & \mid \langle \ell = e \rangle \\ & & & & \mid \mathtt{case}(e, \bar{x}.\bar{e}) \end{array}$$

and the typing rules and evaluation rules are modified by dropping type annotations wherever they appear.

Now the subtyping principles that hold for variants are dual to those that hold for records. We have again depth subtyping for variants:

$$\frac{\tau_i \leq \tau_i' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell} : \bar{\tau}' \rangle}$$

and also subtyping by permutation:

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell_i' \qquad \tau_{\pi i} = \tau_i' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell}' : \bar{\tau}' \rangle}$$

The rule for width subtyping, however, is reversed. While a record with more fields contains more information, (and hence is a subtype of a record with a subset of its fields) a variant with more branches conveys less information: with more possibilities, we are less certain what branch is present. So a variant with more branches is a supertype of a variant type with fewer.

$$\frac{}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell} : \bar{\tau}, \bar{\ell}' : \bar{\tau}' \rangle}$$

By similar reasoning as before, we actually want to combine all three rules into one so that transitivity is admissible instead of necessary as a separate rule. The single rule for variant subtyping is as follows.

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell_i'' \qquad \tau_{\pi i} \leq \tau_i'' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell}'' : \bar{\tau}'', \bar{\ell}' : \bar{\tau}' \rangle}$$

Lecture:
EML and Multimethods

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 4, 2004

1 Object-Oriented Programming

There are several aspects of the *object-oriented* programming style that a proponent of it might point to as centrally important. It advocates achieving modularity of programs through bundling all of the data and behavior of something that can be thought of as an *object*; it encourages reuse of code through *inheritance*; it makes claims on abstraction by use of *dynamic dispatch*. The way a message sent to an object is handled need not be known by the outside world: the object itself 'knows' how to handle it.

Programmers used to programming in a functional style in type systems that support ML-style modules may well have different ideas about what constitutes a natural organization of code and data, and how best to reuse code. But even without getting into arguments about what 'natural' means, we can at least point to one (somewhat) less controversial idiom that many OO languages easily support, which is not as easily (or at least not in the same way) codable in a language like ML.

2 Extensibility

2.1

The idea in question is *data extensibility*. In, for example, Java, suppose you declare an abstract class and some number of concrete classes.

```
abstract class Exp {
 // return result of substituting v
 // for x in this expression
 Exp subst(Exp v, Int x);
}
...
class Apply extends Exp {
  Apply(Exp e1, Exp e2);
  ... // implement subst
}
...
class Fn extends Exp {
  Fn(Exp e);
  ... // implement subst
}
... // more kinds of Exp
```

Here `Exp` is the abstract class that sits at the top of a part of the class hierarchy that describes the syntax of some language of, perhaps a compiler. It declares methods for operations that the client of this code wants to perform — substitution, for example. To describe the various possible ways of making an `Exp`, we create concrete subclasses of `Exp`, and implement its methods.

That the language supports data extensibility is just the fact that if, at a later time, we decide that we want to expand the old datatype with a new construct, say `IsHalting`, then the changes we need to make to the old code consist of simply adding a new class, with new method implementations:

```
class IsHalting extends Exp {
  IsHalting(Exp e);
  ... // implement subst
}
... // more kinds of Exp
```

The important things to notice here are that (a) the modifications are 'all together' in one place, in one file, and (b) they are not changes to existing library code. As a client of library code that defines the datatype `Exp`, we may extend it by adding new kinds of `Exp`s in our own code.

In contrast, ML (with the exception of the `exn` type) requires that variant datatypes give all of their branches at once, and does not permit extension. Were we to implement the example above in ML, we would start with something like

```
structure Syntax :> SYNTAX =
struct
 datatype exp = Apply of exp * exp
              | Fn of exp
              | ...
 val subst (e : exp, v : exp, x : int) = ...
end
```

To add `IsHalting`, we would have to go in and actually change the original datatype declaration:

```
structure Syntax :> SYNTAX =
struct
 datatype exp = Apply of exp * exp
              | Fn of exp
              | IsHalting
              | ...
 val subst (e : exp, v : exp, x : int) = ...
end
```

Moreover we would have to add another case to the function `subst`, and to any other functions that Syntax might define. Even worse, there might be functions that take arguments of type `exp` outside the `Syntax` structure as well! The changes required to our code could be wildly discontiguous, spanning many files. Depending on how much attention the programmer (or her coworkers) pay to eliminating all nonexhaustive match warnings from a programming project, it may be quite difficult to make sure all functions have been extended appropriately.

2.2

Java programmers can take data extensibility for granted, while ML programmers find workarounds, or else just tolerate changing datatype declarations when they must, and hunting down function cases to extend. But

there is a sort of extensibility that ML hackers take for granted that Java hackers symmetrically must go to some pains to acheive: *functional extensibility*. If instead of adding new sorts of data to an existing program what we want to do is add new behavior to existing datatypes, then the difficulty of the task depends on what tools the languages gives us.

Suppose for definiteness that we want to take our compiler above and add an interpreter to it. In the ML case, we're perfectly capable of writing a separate module with a function `eval` like so:

```
structure Eval :> EVAL = struct
 val eval (e : exp) = case e of
                        (Apply(e1,e2)) => ...
                      | (...) => ...
end
```

Note again the advantages we have here: (a) the modifications are 'all together' in one place, in one file, and (b) they are not changes to existing library code. If we write a case for every branch of the type `exp`, then the function works on any `exp` that comes its way. The compiler can provide accurate warnings as to whether our case analysis is nonexhaustive, redundant, or correct.

If we want to do the same thing in Java, then we have at least two obvious options, neither necessarily pleasant. One works only if we have access to the original library, or to its authors. We can add ourselves, or beg the authors to add, a new method to the superclass `Exp`, and implement it in all of the subclasses. This means making many changes in many scattered places, and it is vulnerable to certain kinds of errors. We may implement the method for a subclass C of `Exp` and forget to implement it for a subclass D of C: if the inherited code is not appropriate for D, then we have failed to make enough changes, but there is no way the compiler can tell us. If a library's API changes because at one of its users' behest, other users may be suddenly stuck with broken code because they don't implement the newly added methods to the abstract superclass.

The other apparent option is writing a static function which contains a big `if-elseif-else` full of `instanceOf` tests. Here we are again capable of leaving out cases in ways the compiler can't detect and sentencing ourselves to unexpected runtime errors. Also, in the pursuit of functional extensibility, we've thrown out convenient data extensibility, at least for the purpose of this one new function. For if we create a new class later, we cannot implement its `eval` case as a method (nor will the compiler know that

we should implement `eval` for it all!) but instead we must hunt down the mass of `instanceOf` tests in the static `eval` function and add a new case.

3 EML

3.1

ML beats Java on functional extensibility, and Java beats ML on data extensibility. Can't we all just get along, somehow? EML is an attempt to compromise and get both kinds of extensibility at once. It is, in a sense, both a functional language and an object-oriented language. We're going to discuss a simplified version here, but if you want to read the original paper, you can find it at

`http://www.cs.ucla.edu/todd/research/icfp02.html`

Before we get to how EML supports extensibility, we have to discuss one other feature, since it is necessary to understand many aspects of EML's syntax and semantics. EML, as an object-oriented language, has a feature that is not present in Java, but which is present in quite a few other OO languages, called *multimethods*. It is a natural generalization of the dynamic dispatch found in Java. Where a method call `e1.subst(e2,x)` in Java dispatches on the run-time tag of only one of its inputs, namely `e1`, a method call in a language with multimethods can dispatch on the tags of a tuple of arguments all at once. This would be useful if we wanted substitution to do something different depending on what kind of expression `e2` was.

Because multimethods allow dispatch over lists of objects instead of just single objects, we can dispense with the idea that a method is called 'on' a single object. Instead the syntax is closer to an ordinary function call in a functional language. The semantics of the call are still drawn, however, from the object-oriented paradigm: which code gets called as a result of the method invocation depends on the run-time tags of the arguments.

3.2

The syntax of the programs in the new language (which again we construe as an extension of MinML) consists of *declarations* in addition to expressions and values. We also have a notion of *classes* in addition to *types*. As with record field labels ℓ, we assume there to be infinite supplies of class names C and method named m. We have that every class is a type, but not every

type is a class: we still have all of our old types like `int` and $\tau_1 \times \tau_2$ and so on. Among classes, there is a notion of *subclassing*, which is determined by which classes are actually declared by the programmer to extend one another.

We have new expressions for constructing objects, which are represented as tagged records $\{C : \bar{\ell} = \bar{e}\}$ with tag C and fields $\bar{e}$, a deconstructor for projecting out object fields (as we did with records) and a way to call methods. Formally the language of classes, types, and expressions is extended as follows:

$$
\begin{array}{rlll}
\textit{Classes} & C & & \\
\textit{Types} & \tau & ::= & \cdots \mid C \\
\textit{Values} & v & ::= & \cdots \mid \{C : \bar{\ell} = \bar{v}\} \\
\textit{Expressions} & e & ::= & \cdots \mid \{C : \bar{\ell} = \bar{e}\} \\
 & & & \mid \#\ell(e) \\
 & & & \mid \texttt{call}\, m(\bar{e})
\end{array}
$$

with the usual conventions about $\bar{\ell} = \bar{e}$, etc.

Now the declarations of the language allow us to create classes, create methods, and implement (i.e. 'extend') methods:

$$
\begin{array}{rlll}
\textit{Declarations} & d & ::= & \cdots \mid [\texttt{abstract}]\, \texttt{class}\, C\, [\texttt{extends}\, C'] \\
 & & & \quad \texttt{of}\, \{\bar{\ell} : \bar{\tau}\} \\
 & & & \mid \texttt{method}\, m(\bar{C}) : \tau \\
 & & & \mid \texttt{extend}\, m(\bar{x} : \bar{C}) = e
\end{array}
$$

A declared class may or may not be *abstract* (i.e. disallow instantiation of itself, and only allow subclassing) may or may not extend (i.e. declare itself a subclass of) another class, and has a set of *fields* $\bar{\ell}$ with types $\bar{\tau}$. A method has a tuple of argument classes $\bar{C}$, and a return type τ. When we implement a case of a method — that is, a piece of code that may run when the method is invoked, depending on the run-time tags of the arguments given — we specify names and types for all of its arguments $\bar{x}$, and provide a function body e in which all the variables $\bar{x}$ are bound.

A program in EML consists of a list of *modules* and an expression to be evaluated. A module for our purposes is just a container for a group of declarations. It is something of the following form:

```
module
 decl
 decl
 ...
 decl
end
```

3.3

Before we get to the typing and evaluation rules, let's just take a quick look at some EML code that goes precisely where neither ML nor Java dare tread: pulling off data and functional extensibility at the same time.

Remember we started with a datatype that represented expressions, and a function that performed substitution. In EML we would write this as a module

```
module
 abstract class Exp of {}
 class Apply extends Exp of {e1 : Exp, e2 : Exp}
 class Var extends Exp of {n : Int}
 class Fn extends Exp of {body : Exp}
 ...

 method subst(Exp, Exp) : int -> Exp
// call subst(e, v) x ==> {v/x} e

 extend subst(e : Apply, v : Exp) = fn x =>
 {Apply: e1 = call subst(#e1 e, v) x,
         e2 = call subst(#e2 e, v) x}

 extend subst(e : Var, v : Exp) = fn x =>
   if #n e = x then v else e

 extend subst(e : Fn, v : Exp) = fn x =>
  {Fn: body = call subst(#body e) (x+1)}
 ...
end
```

Just like in Java, we have a abstract (uninstantiable) class of expressions, and one concrete class for every variety of expression. We declare a method that takes two `Exp` arguments, and returns a function that takes an int and returns an `Exp`. We might also view this as a partially curried function taking three arguments. We implement the method by giving its behavior on particular cases of its arguments' run-time tags. If the first argument happens to be an `Apply` and the second (as it will inevitably be, as long as the program is well-typed) is an `Exp`, then the first piece of code will execute. Similarly if the first argument is a `Fn` then the second piece of code will execute.

If we want data extensibility, then in another module we can write

```
module
 class IsHalting extends Exp of {body : Exp}

 extend subst(e : IsHalting, v : Exp) = fn x =>
 ...
end
```

If we want functional extensibility, then in another module we can write

```
module
 method step(Exp) : Exp

 extend step(e : Apply) = ...
 extend step(e : Fn) = ...
 extend step(e : Var) = raise Stuck
 ...
end
```

And if we want both extensions at once, we can write a fourth module completely separate from the above three that fills in the evaluation case for `isHalting`:

```
module
 extend step(e : isHalting) = ...
end
```

From the point of view of ML, what EML gives you is the ability to add new datatypes and new function cases at places in your code far away from the original declarations of those datatypes and functions. From the point of view of Java, what EML allows is the addition of new methods to old class hierarchies (again, far away from the original codebase) in a way that fits smoothly with the existing dispatch mechanism.

3.4

For the typing rules, we assume $Decls$ to be the set of existing declarations. Every time a module is encountered and determined to be well-formed, its declarations are imperatively added to the set $Decls$. For now, well-formedness of modules will depend on all of the previous parts of the program up to that point. In next lecture we will discuss how to make these checks modular, i.e. local to the current module.

An object expression is well-typed if its tag represents a declared class, and its fields are all well-typed.

$$\frac{\texttt{class}\ C\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls \qquad \Gamma \vdash e_i : \tau_i \quad \text{(for all } i\text{)}}{\Gamma \vdash \{C : \bar{\ell} = \bar{e}\} : C}$$

If the class C subclasses another class C', then the left-over fields must form a valid object of class C'.

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls \qquad \begin{array}{c}\Gamma \vdash e_i : \tau_i \quad \text{(for all } i\text{)} \\ \Gamma \vdash \{C' : \bar{\ell}' = \bar{e}'\} : C'\end{array}}{\Gamma \vdash \{C : \bar{\ell} = \bar{e}, \bar{\ell}' = \bar{e}'\} : C}$$

A projection expression is well-typed if its body is an object of a class with the indicated field.

$$\frac{\ell : \tau \in Fields(C) \qquad \Gamma \vdash e : C}{\Gamma \vdash \#\ell(e)}$$

where $Fields$ is defined by

$$\frac{\texttt{class}\ C\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{Fields(C) = (\bar{\ell} : \bar{\tau})}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{Fields(C) = (\bar{\ell} : \bar{\tau}, Fields(C'))}$$

A method call is well-typed as long as all of its arguments are:

$$\frac{\texttt{method}\ m(\bar{C}) : \tau \in Decls \qquad \Gamma \vdash e_i : C'_i \quad \text{(for all } i\text{)} \quad \bar{C}' \leq \bar{C}}{\texttt{call}\ m(\bar{e}) : \tau}$$

where here $\leq$ is the declared subclass relation defined by

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{of}\ \{\bar{\ell} : \bar{\tau}\} \in Decls}{C \leq C'}$$

$$\frac{}{C \leq C}$$

$$\frac{C \leq C' \qquad C' \leq C''}{C \leq C''}$$

Since there are only finitely many classes declared in a given program, introducing a transitivity rule here is not harmful. We can efficiently compute the reflexive, transitive closure of the is-a-direct-subclass relation. The notation $\bar{C} \leq \bar{C}'$ means that $C_i \leq C'_i$ for all i.

3.5

The operational semantics of EML depend on the idea of a *best match* for a given method call. Suppose we have an object hierarchy consisting of the classes Square and Rect, where Square is a subclass of Rect, and the method `Intersect(Rect, Rect):bool`. If we declare cases for Intersect for all four possibilities of Square and Rect inputs,

```
...
extend Intersect(Rect, Rect) = ...
extend Intersect(Rect, Square) = ...
extend Intersect(Square, Rect) = ...
extend Intersect(Square, Square) = ...
...
```

then we can consider what happens when we invoke the method by writing `call Intersect(e1, e2)` for some expressions `e1, e2`. If `e1` and `e2` are both Squares, then although all four cases match — in the sense that

the arguments given by the method call are subtypes of the arguments required by each case — only the fourth case seems appropriate, since it is the most specific.

We don't want to require exact matches between the run-time arguments and the case's arguments, for if we only wrote

```
...
extend Intersect(Rect, Rect) = ...
...
```

we would still like the code to work on Squares as well by inheritance. So *if* there is a more specific case we will use it, but any maximally specific case that matches is the one we will execute.

Formally, the evaluation rules are comprised of search rules

$$\frac{e \mapsto e'}{\{C : \bar{\ell}^\flat = \bar{v}^\flat, \ell = e, \bar{\ell}^\sharp = \bar{e}^\sharp\} \mapsto \{C : \bar{\ell}^\flat = \bar{v}^\flat, \ell = e', \bar{\ell}^\sharp = \bar{e}^\sharp\}}$$

$$\frac{e \mapsto e'}{\#\ell(e) \mapsto \#\ell(e')} \quad \frac{e \mapsto e'}{\texttt{call}\ m(\bar{v}, e, \bar{e}) \mapsto \texttt{call}\ m(\bar{v}, e', \bar{e})}$$

the reduction rule

$$\frac{}{\#\ell_i(\{C : \bar{\ell} = \bar{v}\} \mapsto v_i}$$

and the dispatch rule

$$\frac{\begin{array}{c}\texttt{extend}\ m(\bar{C}') = e \in Decls \\ \bar{C} \leq \bar{C}' \\ \text{For any other } \texttt{extend}\ m(\bar{C}'') = e' \in Decls \text{ such that} \\ \bar{C} \leq \bar{C}'' \text{ we have } \bar{C}' \leq \bar{C}''\end{array}}{\texttt{call}\ m(v_1\ \texttt{as}\ \{C_1 : \cdots\}, \ldots, v_n\ \texttt{as}\ \{C_n : \cdots\}) \mapsto \{\bar{v}/\bar{x}\}e}$$

This last rule expresses that we dispatch to a method case if (a) the arguments $\bar{C}$ are individually subclasses of the required arguments $\bar{C}'$ of that case and (b) any other case $m(\bar{C}'')$ that also matches the arguments $\bar{C}$ is more general (i.e. less specific) than this case.

3.6

The problem with the operational semantics given is that it may get stuck, in one of two ways.

One is that we try invoking a method for which no case is applicable. This happens if we have declared the Intersect method and implemented it only for a pair of Squares, and try to invoke it on a pair of Rectangles (or on a Rectangle and a Square).

The other is that more than one case is applicable, and no single case is the most specific. If what we have implemented is only

```
...
extend Intersect(Rect, Square) = ...
extend Intersect(Square, Rect) = ...
...
```

and we try to invoke Intersect on two Squares, then both cases are applicable, but neither is more specific than the other. It would take an implementation of

```
extend Intersect(Square, Square) = ...
```

to remedy this.

In the next lecture we will discuss EML's method for preventing these kind of run-time errors, in an efficient and local way. Until then, imagine that the type-checker does a global pass over the program, trying by brute force to determine whether these exhaustivity and ambiguity errors can arise.

For exhaustivity, we can simply enumerate all of the possible argument lists that could legitimately be given to a declared method, and check for the existence of a case that handles each one.

For ambiguity, we can, for each method, enumerate every pair of declared cases it has. Suppose the case we have in mind are `extend` $m(\bar{C}) = e$ and `extend` $m(\bar{C}') = e'$. Now for a set of arguments to cause an ambiguity error to occur with these two cases, it would have to match both of them. That is, there would be a list of classes $\bar{C}''$ such that $\bar{C}'' \leq \bar{C}$ and $\bar{C}'' \leq \bar{C}'$. For each position in this list, if there are any common subclasses of C_i and C_i', there is a 'most super' subclass, a subclass highest in the hierarchy. In fact it must be either C_i or C_i'. This follows from the fact that we have no

multiple inheritance. Finding this 'most super' subclass is also called finding the *greatest lower bound* of C_i and C_i', which we can write as $C_i \cap C_i'$. It may be, however, if neither C_i nor C_i' is a descendant of the other, that there is no lower bound at all. If this is ever the case, we are safe with respect to this pair of cases, for no list of arguments could ever satisfy both.

Otherwise we can do this for all elements in the list, thus finding $\bar{C} \cap \bar{C}'$. This list of classes, if it exists, is the most general set of arguments that is specific enough to possibly trigger the two cases we began considering. Now we simply check whether $\bar{C} \leq \bar{C}'$ or $\bar{C}' \leq \bar{C}$. If either of these holds, we are safe, for there is no ambiguity. One case is more specific than the other. Otherwise, we throw up a compile-time error, because invoking method m with arguments tagged with $\bar{C} \cap \bar{C}'$ would cause a run-time ambiguity error.

Lecture:
EML and Multimethods

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 9, 2004

1 Safety in EML

Last time we discussed two kinds of errors that an ill-formed EML program might produce at runtime. We'll refer to them as:

1. *Message not found*: a method call ('message') made by the program 'can't find a match,' i.e. can't find a method implementation corresponding to the arguments given.

2. *Message ambiguous*: a method call made by the program can't find a *unique* match, i.e. some method implementation cases match, but there is no most specific case.

We want to pick restrictions we can put on programs at compile-time so that these errors cannot occur at run-time. In the previous lecture, we simply ran a global analysis to see by brute force that every possible set of arguments that we might call a method with in fact determined a unique case. In a large program with a lot of classes and methods and method cases, this could be very expensive.

So what we'd like to do — and this is what makes EML interesting as compared to a naïve multimethod language — is do some compile-time checks to prevent these errors in a *modular* way. That is, we check the validity of a program one unit at a time. The unit of checking is going to be exactly the `module` $\cdots$ `end` blocks that we introduced already.

What it means to perform a 'modular' check on a module is that we *only depend on* which classes, methods, and method extentions have been declared in modules that the current module *statically depends* on, in the

following sense: we say the *interface* of a module is the set of classes, methods, and method exentions it declares, ignoring the actual implementation of the method extentions. A module M is then said to statically depend on another module M' if M's interface mentions a name (i.e. class name or method name) defined in the interface of M'. Also, if M_1 depends on M_2 which depends on $\cdots$ which depends on M_n, then we say that M_1 depends on M_n.

At the end of the day, when we check a module, we need only consider the current module and the interface of all the modules it depends on. This is similar in spirit to the encapsulation that Java and ML provide: you depend on a class's interface, or a structure's signature, but not the class's or the structure's implementation.

1.1 Completeness of Methods

We'll deal with *Message not found* first. Think about how languages like Java and ML deal with this issue.

In ML, it does appear at run-time as the error message `uncaught exception nonexhaustive match failure`, but the compiler also produces a warning message, `Warning: match nonexhaustive` when a function with missing cases is written. How the compiler knows that cases are actually missing involves the fact that, in an ML program, all of the cases of a function must appear as part of the function's definition. This fact means that the inherently local analysis the compiler does — that is, look at all of the cases given in the function definition and see if they're sufficient to cover any possible input — is correct.

In Java, the compiler can prevent a method from being invoked on objects that don't know how to implement it, because among other reasons, it knows what type the object has at run-time. If I try to downcast an `Object` to a `FloorWaxer` and call the method `waxFloor()` on it, the exception is raised at the downcast, not at the method call. Moreover, the compiler guarantees that any subclass of `FloorWaxer` must either inherit an implementation of or give its own implementation of all methods, e.g. `waxFloor()`. Because of this, if we ever have an expression with type `FloorWaxer`, we know it is safe to invoke `waxFloor()` on it, because it is statically guaranteed to be of some class that implements that method.

In this simplified version EML we will take a cue from Java and treat the first argument to each EML method as somewhat special.[1] This special

[1]The real EML allows the programmer to designate any of a tuple of arguments as this

treatment in no way changes our existing dispatch method: dispatch is still simultaneous and symmetric across all method arguments. We treat the first argument specially only for purposes of the compile-time restrictions we're about to introduce.

To prevent method calls from producing 'message not found' errors, informally we say the following: *every method must have at least as many implementations as an ML function or a Java method*. It may also have more.

The way we decide whether a method is 'more ML-like' or 'more Java-like' is based on the the first argument. If our code looks like

```
module
 class C of ...
 method m(C,D) : bool
 ...
end
```

then the method `m` occurs in the same module as the declaration of the class of its first argument. We call such methods **internal** methods. Since declaration of new methods in Java always occurs in the same file as the declaration of the class of their (implicit) first argument (i.e. `this`) we will treat internal methods similarly to OO-style methods.

> **Restriction 1**: *Internal Methods Require Local Defaults*. Suppose there is an internal method m with arguments $C, D_1, \ldots, D_n$. If C is abstract, then for any concrete subclass C' of C, there must be a *local default case for m and C'*, that is, a case `extend m(`$C', D_1, \ldots, D_n$`) = e` in the same module as the declaration of C'.

This restriction is checked whenever a concrete subclass of another class is declared. When this happens, we look to which methods are declared whose first argument is one of our superclasses. If we cannot inherit an implementation from a superclass, then we must write a local default case.

The other restriction applies to methods that are *not* declared in the same module as their first argument's class. These methods are called **external**. For these we need to impose a stronger condition.

---

singled-out 'owner' position.

Restriction 2: *External Methods Require Global Defaults*. Suppose there is an external method m with arguments $C, D_1, \ldots, D_n$. Then in the same module as m there must be a *global default case* for m, that is, a case `extend m(`$C, D_1, \ldots, D_n$`) = e`

Here we simply guarantee that the most general case of the method is covered, and therefore trivially no 'message not found' error can occur.

By fairly simple reasoning we can see that these two restrictions are sufficient for any well-typed program. For every method call that occurs at run-time, either the method is internal or external. If it is external, then somewhere there has been defined a global default, so we are covered. If it is internal, then since we assumed the program was well-typed, we know the argument given has a run-time tag that is a subclass C' of the first argument C of the method being called. This means that in some module, we declared the class C', as a (possibly indirect) subclass of C. At that time, by restriction 1, we must have checked that a local default case for C' existed in the same module. This local default case is sufficient to show that at least one applicable case matches our method call, QED.

Notice that although we require a sort of imitation of either OOP languages in requiring local defaults or FP in requiring that we are certain of exhaustiveness at the point of declaration of a method (in this case by requiring a global default) this imitation is only a minimum requirement: we can also define more specific cases in addition to the local or global defaults for either internal or external methods, and this is a strict improvement over what ML or Java offers us.

1.2 Nonambiguity of Methods

To prevent ambiguous message warnings, we similarly turn to familiar language paradigms for ideas.

The reason Java doesn't have ambiguity problems is that although it does OO-style dynamic dispatch, it's only *single* dispatch. The run-time dispatch that takes place during the call `x.method(y,z)` only depends on the run-time tag of `x`, not those of `y,z`. This is related to the fact that all method *implementations* are textually bundled up with the class of their first argument.[2] The only place you can override a method — and what overriding a method means is creating a new specialized behavior for when the

[2]Be careful not to confuse this with the idea in the definition of internal vs. external: that was about method *declarations* being bundled with the class of their first argument

reciever of a method call is some class C — is in the scope of the declaration of class C.

One reason that means already that ML can't have ambiguity problems is that its semantics for case analysis is different from EML's. In ML, if multiple cases match, then the earlier cases have precedence. However, even if ML had EML's semantics of preferring the most specific case regardless of order (and the potential for ambiguity that comes with it) it could still do a good job of warning about ambiguity and compile-time. This is because all cases of a case analysis have to occur all in the same place: in EML terms, all of the method implementations have to be in the same place as the method declaration.

We don't wish to impose either of these restrictions wholesale on the programmer, so we give them the choice extention-by-extention whether to make a particular case (a) OOP-style (keeping it with the class of its first argument) or (b) FP-style (keeping it with the method declaration).

> **Restriction 3**: *Nonambiguity Constraint*. Every method implementation `extend m(`$C, D_1, \ldots, D_n$`) = e` must occur either (a) in the same module as the declaration of C or (b) in the same module as the declaration of m

Why does this prevent ambiguous match errors at run-type? Here is a sketch of a proof. Suppose we have a method call `call m(`$e_1, \ldots, e_n$`)`, and e_1 evaluates to an object value $\{D : \cdots\}$ of class D. Suppose that two cases match this call, say, `extend m(`C'`, ... ) =` e' and `extend m(`C''`, ... ) =` e''. In particular we know D is a subclass of both C' and C''.

Recall that, when checking the validity of a module, we do have available all the information in the interfaces of all the modules that module depends on. This means that if either of these two extentions satisfied part (b) of the nonambiguity contraint above, then any potential ambiguity would have been detected by the compiler. For suppose without loss of generality the first case `extend m(`C'`, ... ) =` e' is declared in the same module as the method m itself. Then the second case, `extend m(`C''`, ... ) =` e'', since it mentions the method m, statically depends on that module. So when we check the module containing the second case, the first case will be visible when we do the ordinary ambiguity checks mentioned in last lecture.

If both extentions satisfy only part (a) of the constraint, however, we have to reason differently. In that case it may be that the two cases occur in different modules, say M' and M'', each of which statically depends on the module that declares m, but neither of M',M'' depends on the other. (If

it happens that one *does* depend on the other, we are already done, for the same reason as the last case: suppose it's M' that depends on M''. While checking M' we will 'see' M'' and therefore see both extentions and detect that they are ambiguous) But we must actually satisfy part (a) for each extention. This means that C' is declared in M' and C'' is declared in M''. Because M' and M'' are assumed not to depend on one another, this means neither of C' and C'' are subclasses of one another. Since we don't have multiple inheritance, this means the fact that D is a subclass of C' and C'' is a contradiction, QED.

Lecture Notes on Storage Management

15-312: Foundations of Programming Languages
Daniel Spoonhower
Modified by Frank Pfenning

Lecture 21
November 11, 2004

In our discussion of mutable storage, a question was raised: if we allocate a new storage cell for each `ref` expression we encounter, when do we release these storage cells? As we will discover today, a similar question will be raised when we reconsider our implementation of pairs, lists, and closures, or generally any aggregate data structure.

In designing the `E` machine, our goal was to describe a machine that more accurately modeled the way that programs are executed on real hardware (for example, by using environments rather than substitution). However, most real machines will treat *small* values (such as integers) differently from *large* values (such as pairs and closures). Small values may be stored in registers or on the stack, while larger values, such as pairs and closures, must be allocated from the *heap*. While the storage associated with registers and the stack can be reclaimed at the end of a function invocation or lexical scope, there is no "obvious" program point at which we can reuse the storage allocated from the heap.

Clearly, for programs that run for hours, days, or weeks, we must periodically reclaim any unused storage. One possible solution is to require the programmer to explicitly manage storage, as one might in languages such as C or C++. However, doing so not only exposes the programmer to a host of new programming errors, but also makes it exceedingly difficult to prove properties of languages such as preservation.

An alternative approach is to require that the implementation of the language manage storage *for* the programmer. Automatic memory management or *garbage collection* can be found in most modern languages, in-

cluding Java, C#, Haskell, and SML.

In this lecture, we will modify and extend the semantics of the E machine to account for the differences between small and large values and include new transition rules for automatically reclaiming unused storage.

The A Machine

In order to extend the semantics of the E machine with transition rules for automatic storage management, we must enrich our model of expressions, values, and program states. For the purposes of our discussion today, we will use a version of MinML that includes integers, functions, and lists. As we alluded to above, in order to provide a framework for automatic storage management, the A machine will distinguish *small* values from *large* values, as follows.

$$\begin{array}{llll} \textit{Small Values} & v & ::= & \texttt{num}(n) \mid \texttt{nil} \\ \textit{Large Values} & w & ::= & \langle\!\langle \eta; e \rangle\!\rangle \mid \texttt{cons}(v_1, v_2) \end{array}$$

Closures and cons cells (*i.e.* large values) will not be stored directly in the stack or environment; instead we will use *locations* to refer to them indirectly. As in our formulation of references, locations (denoted syntactically as l) will not appear in the concrete syntax.

$$\begin{array}{llll} \textit{Locations} & l & & \\ \textit{Expressions} & e & ::= & \ldots \mid \texttt{loc}(l) \\ \textit{Small Values} & v & ::= & \ldots \mid \texttt{loc}(l) \end{array}$$

We will also maintain a finite mapping from locations to large values, called a *heap*. We allow locations to appear in the stack and environment, but whenever we are forced to compute with a pair or closure, we must look-up the actual value in the heap.[1]

$$\begin{array}{llll} \textit{Heaps} & H & ::= & \cdot \mid H, l{=}w \\ \textit{Environments} & \eta & ::= & \cdot \mid \eta, x{=}v \\ \textit{States} & s & ::= & H \;;\; k > e \\ & & \mid & H \;;\; k < v \end{array}$$

Frames f and stacks k are given as before but with the replacement of small values for values.

[1]The heap is similar in notion to the *store* as it appeared in our discussion of mutable references; however, while the store may be updated by assignment, the heap is immutable from the programmer's perspective.

Since the A machine does not allow small values to be maintained in or returned to the stack, in states where we previously returned large values, we must instead create and look-up locations. For example, cons cells are now introduced and eliminated according to the following rules.

$$H\ ;\ k > \texttt{cons}(e_1, e_2) \mapsto_{\texttt{a}} H\ ;\ k \triangleright \texttt{cons}(\Box, e_2) > e_1$$

$$H\ ;\ k \triangleright \texttt{cons}(\Box, e_2) < v_1 \mapsto_{\texttt{a}} H\ ;\ k \triangleright \texttt{cons}(v_1, \Box) > e_2$$

$$H\ ;\ k \triangleright \texttt{cons}(v_1, \Box) < v_2 \mapsto_{\texttt{a}} H, l{=}\texttt{cons}(v_1, v_2)\ ;\ k < \texttt{loc}(l)$$

$$H\ ;\ k > \texttt{case}(e_1, e_2, x.y.e_3) \mapsto_{\texttt{a}} H\ ;\ k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) > e_1$$

$$H\ ;\ k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) < \texttt{nil} \mapsto_{\texttt{a}} H\ ;\ k > e_2$$

$$H\ ;\ k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) < \texttt{loc}(l) \mapsto_{\texttt{a}} H\ ;\ k \blacktriangleright (x{=}v_1, y{=}v_2) > e_3$$

where $l{=}\texttt{cons}(v_1, v_2)$ in H

Recall that environment frames $k \blacktriangleright \eta$ on the stack are popped when values are returned past them, and that variables are looked up in the environments on the stack from right to left (see also Assignment 4 and the code in the sample solution). We will now return to the question, when can values safely be removed from the heap?[2]

Garbage and Collection

We would like to state that "the collector does not change the behavior of the program." That is, garbage should be exactly those parts of the program state that do not affect the result of evaluation. Consider the following program,

```
(let p = cons(3,cons(4,nil)) in
  case p of nil => 2
     | cons(n,k) = p in
    [a] fn x => n
  end
end [b]) 7 [c];
```

If we allocate `p` as described above, when it is safe to free it? At point $[a]$? $[b]$? $[c]$? We would like to release the storage associated with a location

[2]Though if we recall our original question with respect to references, we should note that the ideas described here can also be extended to encompass mutable storage.

as soon as it becomes unnecessary to the correct execution of the program. As it turns out, we will not be able to determine exactly when a particular location is no longer necessary: doing so is undecidable!

Instead we will make a conservative[3] assumption about whether or not a location is necessary: we will assume that any location that is *reachable* may be necessary. To do so, we will need to enumerate the *free locations* of a heap, stack, environment or value. (For the moment will we use the syntax $FL()$ to informally refer to these free locations; we will be more precise later.)

Given this notion of garbage, collection is exactly the process of removing garbage from the heap. During our discussion of mutable storage, something akin to the following transition rule was suggested.

$$\frac{FL(H, k, \eta) = \emptyset}{H \cup H' \ ; \ k > e \mapsto_{\mathsf{a}} H \ ; \ k > e} \ \textbf{?}$$

Recall that this rule was deficient in its inability to reclaim (unreachable) cycles in H. For the time being, however, we will tackle a larger problem: how can we separate H from H'?

Tracing Collection

At the most abstract level, the garbage collector has to traverse the stack k and follow chains of location pointers in the heap in order to see which locations may still be relevant to the evaluation of e in k. Note that an expression e may contain free variables (which will be bound to small values in environment in k), but never free locations. This means we don't have to traverse e to see which heap cells may be "live" for the current computation. This general technique is called *tracing*. We now describe a tracing collector using our notation of judgments. In what follows we describe more concrete realizations of this general idea that are closer to what actual implementations do.

The state of the garbage collector has the form $H_f \ ; \ k \ ; \ H_t$ where H_f is the so-called *from-space* that we are traversing and H_t is the so-called *to-space* where we move reachable locations found in H_f. Since locations remain abstract, we simply move them from H_f to H_t. The judgment above is invoked in the following way:

[3] "Conservative" is also, somewhat erroneously, used to describe garbage collection in the presence of incomplete knowledge of the structure of the stack or heap (*e.g.* as in an implementation of C).

$$\frac{H\ ;\ k\ ;\ \cdot \mapsto_{\mathsf{g}}^{*} H_f\ ;\ \bullet\ ;\ H'}{H\ ;\ k > e \mapsto_{\mathsf{a}} H'\ ;\ k > e}$$

That is, we start the garbage collector with the current heap H as the from-space and an empty to-space. Then we trace k and H, moving locations to the to-space until the stack is empty and we can return to the normal evaluation.

Note that this rule can apply whenever we are in the process of evaluating an expression. In a more realistic scenario the garbage collector either starts when we run out of space or acts concurrently on the heap.

Next we describe the rules for garbage collection, using single-step transitions. We use the stack k as a "stack", pushing onto it those portions of the small values that we may still have to trace. Since a stack cannot have values on it directly, only environments, we will use environment with anonymous variables. Recall the invariants on expressions (only free variables, no locations), environments (binds variable to small values) and heaps (binds locations to large values).

$$\begin{array}{lll}
H_f\ ;\ k \triangleright \mathtt{cons}(\Box, e_2)\ ;\ H_t & \mapsto_{\mathsf{g}} & H_f\ ;\ k\ ;\ H_t \\
H_f\ ;\ k \triangleright \mathtt{cons}(v_1, \Box)\ ;\ H_t & \mapsto_{\mathsf{g}} & H_f\ ;\ k \blacktriangleright (_{=}v_1)\ ;\ H_t \\
H_f\ ;\ k \triangleright \mathtt{case}(\Box, e_2, x.y.e_3)\ ;\ H_t & \mapsto_{\mathsf{g}} & H_f\ ;\ k\ ;\ H_t \\
\\
H_f\ ;\ k \blacktriangleright \cdot\ ;\ H_t & \mapsto_{\mathsf{g}} & H_f\ ;\ k\ ;\ H_t \\
H_f\ ;\ k \blacktriangleright (\eta, x{=}\mathtt{nil})\ ;\ H_t & \mapsto_{\mathsf{g}} & H_f\ ;\ k \blacktriangleright \eta\ ;\ H_t \\
(H_f, l{=}\mathtt{cons}(v_1, v_2))\ ;\ k \blacktriangleright (\eta, x{=}l)\ ;\ H_t & \mapsto_{\mathsf{g}} & \\
\qquad H_f\ ;\ k \blacktriangleright (\eta, _{=}v_1, _{=}v_2)\ ;\ H_t, l{=}\mathtt{cons}(v_1, v_2) & & \\
H_f\ ;\ k \blacktriangleright (\eta, x{=}l)\ ;\ (H_t, l{=}w) & \mapsto_{\mathsf{g}} & H_f\ ;\ k \blacktriangleright \eta\ ;\ (H_t, l{=}w)
\end{array}$$

Similar rules apply to closures; some of them are given in Assignment 8 on *Garbage Collection* where more details can be found. Note that the last two rules distinguish the two cases where a heap cell has still to be moved, or has already been moved. In the first case, we push v_1 and v_2 onto the stack, since we have to trace any pointers in them as well. Note that circular data structures, although they cannot be constructed in the given language fragment, present no problem to the garbage collector.

Given our definition of a garbage collector, we could now prove not only that the algorithm terminates, but that it is safe, and it preserves the meaning of programs according to our previous definitions of MinML. The first proof is relatively straightforward; the latter two follow in a manner similar to our proofs for the E machine (with the addition of typing rules for the heap H).

Copying Collection

We now give a slightly lower level view of garbage collection where both from-space and to-space are actually regions in memory whose cells are addressed by integers. In this case, we actually divide the whole available memory into two disjoint regions: one that the evaluator uses, and one that is reserved for the time that we need to call the collector.

Heap cells are allocated from lower to higher addresses, using a special `next` pointer to keep track of the next available address. The garbage collector is invoked when we are attempting to use more than half of the available space.

We then trace the stack and the cells in from-space, moving the cell contents to to-space as we encounter them. Of course, references to memory in the stack need to be updated to point to the new locations of the cells.

Moreoever, we need to account for multiple pointers to the same locations. In order to preserve sharing, we replace the cell content by a *forwarding pointer* that goes from from-space to to-space. When we encounter a forwarding pointer when tracing the heap, we just update the pointer in the stack to the destination of the forwarding pointers.

Once the whole stack has been traced, all reachable cells have been moved to the beginning of the to-space. As this point we flip the roles of the two semi-spaces and resume evaluation.

A pictorial example of copying collection can be found in Figure 1. The contents of blank cells is irrelevant for the purposes of the garbage collection algorithm. They will never be visited because tracing never reaches them.

There are many refinements of copying collection. For example, in order to avoid using additional stack space for tracing, we use a second pointer in to-space so that we always know we still have to trace the region between this second pointer and the `next` pointer. In essence, we use the heap as a kind of special purpose stack.

Other refinements include *incremental collection*, where we do not completely stop the running program but interleave actions of the garbage col-

lector with actions of the running program, and *generational collection* where we collect smaller parts instead the whole semi-space all at once.

Mark-and-Sweep Collection

Another important algorithm for garbage collection is mark-and-sweep, even though it seems to have fallen into disfavor more recently.

A mark-and-sweep collector does not divide the heap into two semi-space, but reserves an additional bit for each heap cell called a mark. Initially all heap cells are unmarked, and the heap is arranged into a linked list of cells called the *free list*. When we allocate an element from the heap we take the first element from the free list and update the free list pointer to its next element.

When the free list become empty, we have to invoke the garbage collector. It traces the heap, starting from the stack, much in the same way as the copying collector. However, rather than copying heap elements it marks them as being reachable.

In a second phase the garbage collector sweeps through the whole memory (not just the reachable cells). During this sweep it adds any unmarked cells to the free list and removes the mark from any marked cells.

A graphic example of mark-and-sweep collection can be found in Figure 2.

In Assignment 8 you have the opportunity to compare copying and mark-and-sweep collection and assess their relative merits, so we will not give a detailed analysis here. One advantage of copying collection that your analysis will probably not be able to reveal is *locality*. When copying, we actually move the elements of the data structures closer together, at the beginning of the to-space. This means better cache behavior which can have dramatic impact on running times on modern machine architectures. As a result, even more mark-and-sweep garbage collectors some algorithm for compacting memory have been developed to avoid the natural *fragmentation* of the heap.

Reference Counting

In a reference counting garbage collector every cell has a counter associated with it that tracks the number of references to it. When we allocate a cell, this counter is initialized to 1. Operations of the (abstract) machine need to maintain these counters. As soon as one of them becomes 0, the cell is

deallocated and the reference counts of the cells that it might point to are decremented, leading perhaps to further garbage collection.

Reference counting is suspect for the heaps of functional languages because of the overhead of maintaining the reference counts, and because it does not work properly with circular structures which prevent reference counts from going to 0! However, the are many less general situations where reference counts are appropriate, such as file descriptors in an operating system, or channels for communication in a distributed environment. In those situations, the overhead of maintaining reference counts is small, while a tracing collector would be hard or impossible to implement because we may not know or even have access to the internals of all processes that my access a resource.

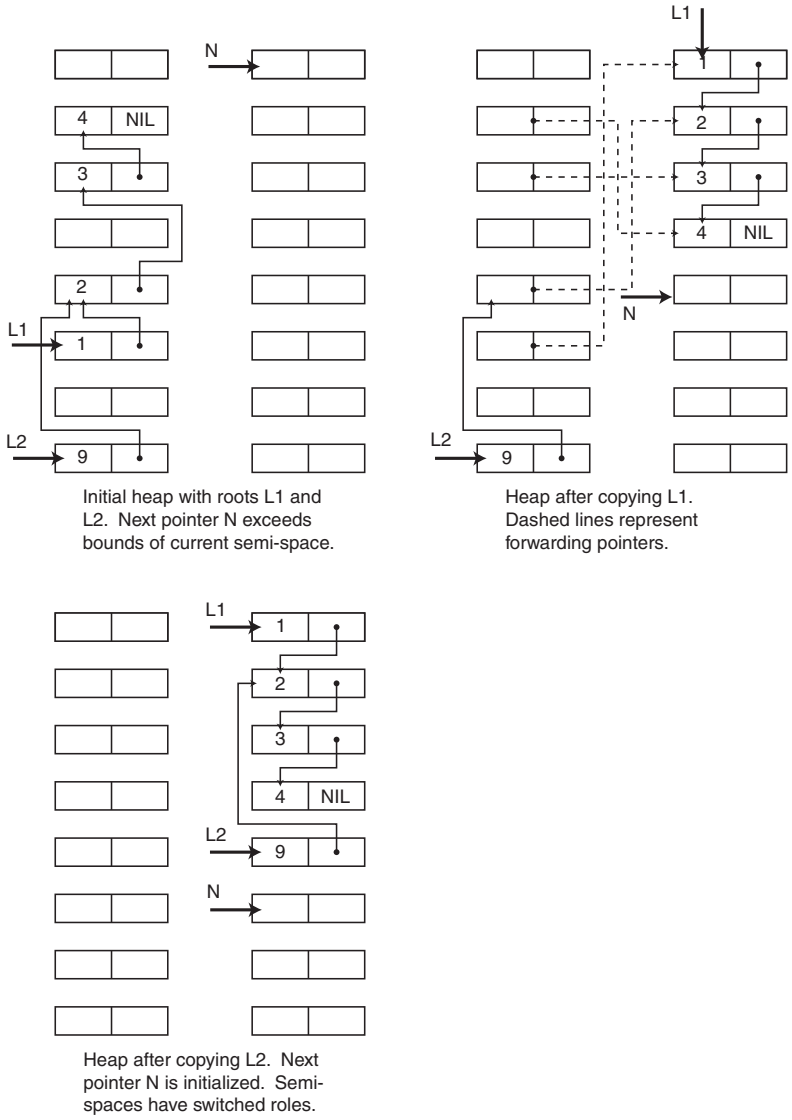


Figure 1: Example of Copying Garbage Collection

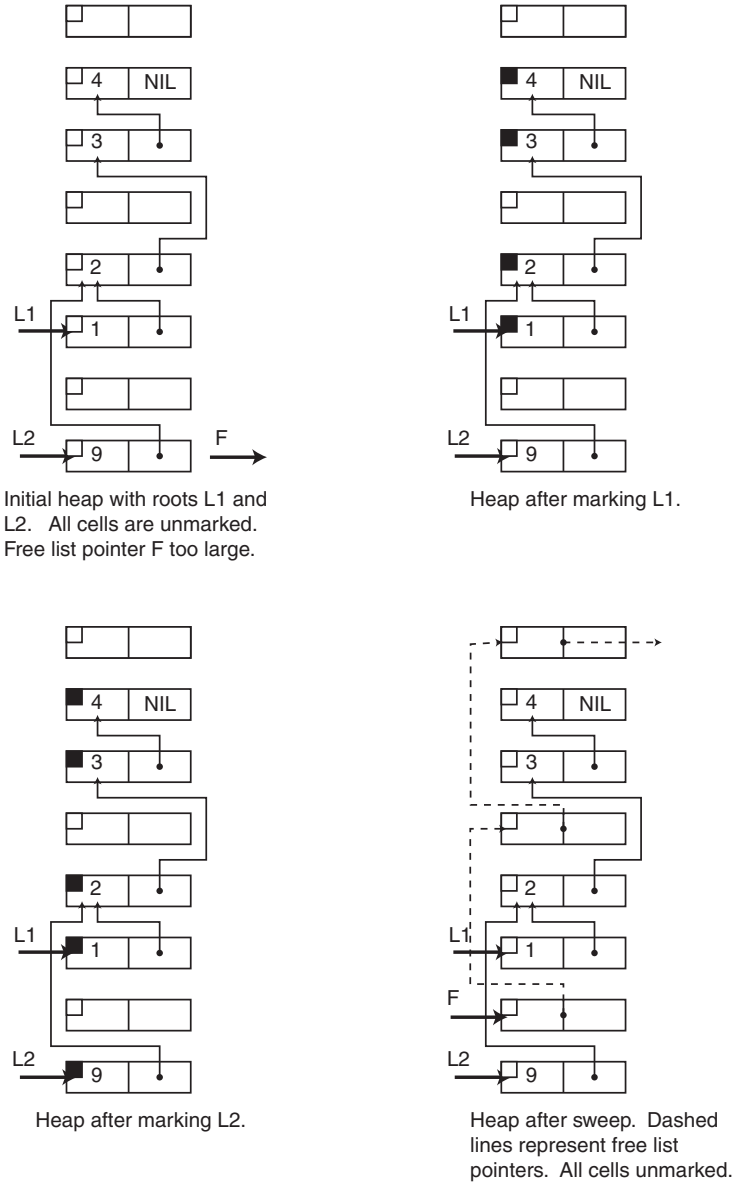


Figure 2: Examples of Mark and Sweep Garbage Collection

Lecture Notes on Call-by-Need and Futures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 22
November 16, 2004

In this lecture we first examine a technique to specify the operational semantics for *call-by-need*, sometimes called *lazy evaluation*. This is an implementation technique for a call-by-name semantics that avoids re-evaluating expressions multiple times by memoizing the result of the first evaluation. Then we use a similar technique to specify the meaning of *futures*, a construct that introduces parallelism into evaluation. Futures were first developed for Multilisp, a dynamically typed, yet statically scoped version of Lisp specifically designed for parallel computation. A standard reference on futures is:

> Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems, 7(4):501-538, October 1985.

One advantage of call-by-name function application over call-by-value is that it avoids the work of evaluating the argument if it is never needed. More broadly, lazy constructors avoid work until the data are actually used. In turn, this has several drawbacks. One of them is that the efficiency model of such a language is more difficult to understand than for a call-by-value language. The second is that lazy constructors introduce infinite values of data types which complicate inductive reasoning about programs. However, the most obvious problem is that if an expression is used several times it will be computed several times unless we can find an implementation technique to avoid this.

There are two basic approaches to avoid re-evaluation of the argument of a function application. The first is to analyze the function body to determine if the argument is really needed. If so, we evaluate it eagerly and

then work with the resulting value. This is semantically transparent, but there are many cases where we cannot tell statically if an argument will be needed. The other is to create a so-called *thunk*[1] and pass a reference to the thunk as the actual argument. When the argument is needed we evaluate the thunk and memoize the resulting value. Further reference to the thunk now just returns the value instead of evaluating it again. Note that this strategy is only a correct implementation of call-by-name if there are no effects in the language (or, if there are effects, they are encapsulated in a monad).

We can think of a thunk as a reference that we can write only once (the first time it is accessed) and henceforth will continue to be the same value. So our semantic specification for call-by-need borrows from the ideas in the operational semantics of mutable references. We generalize the basic judgment $e \mapsto e'$ to $\langle H, e\rangle \mapsto \langle H', e'\rangle$ where H and H' contains all thunks, and e and e' can refer to them by their labels.

$$\textit{Thunks} \qquad H ::= \cdot \mid H, l{=}e$$

Note thunks may be expressions; after they have been evaluated the first time, however, they will be replaced by values. First, the rules for call-by-name application.

$$\frac{\langle H, e_1\rangle \mapsto \langle H', e_1'\rangle}{\langle H, \mathtt{apply}(e_1, e_2)\rangle \mapsto \langle H', \mathtt{apply}(e_1', e_2)\rangle}$$

$$\frac{}{\langle H, \mathtt{apply}(\mathtt{fn}(x.e_1), e_2)\rangle \mapsto \langle (H, l{=}e_2), \{l/x\}e_1\rangle}$$

In the second rule, the label l must be new with respect to H. When the value of l is actually accessed, we need to force the evaluation of the thunk and then record that value.

$$\frac{\langle (H_1, l{=}e, H_2), e\rangle \mapsto \langle (H_1', l{=}e^*, H_2'), e'\rangle}{\langle (H_1, l{=}e, H_2), l\rangle \mapsto \langle (H_1', l{=}e', H_2'), l\rangle}$$

$$\frac{v \ \ \mathsf{value}}{\langle (H_1, l{=}v, H_2), l\rangle \mapsto \langle (H_1, l{=}v, H_2), v\rangle}$$

Note that in the first rule, the result e^* must actually be equal to e. If it were not, that means the evaluation of e would actually require the thunk

[1] The name is a whimsical past tense of *think* derived from "something that has been thought of before".

l, which would lead to an infinite loop. This particular form of infinite loop is called a *black hole* can be detected, while other forms of non-termination remain.

It is left as an exercise to extend the statements of progress and preservation, or to show in which sense the call-by-name semantics coincides with the call-by-need semantics. Note also that there are other rules that can create thunks: essentially every time we need to substitute for a variable. We show one of these cases, namely recursion.

$$\frac{}{\langle H, \mathtt{rec}(x.e)\rangle \mapsto \langle (H, l{=}\{l/x\}e), l\rangle}$$

As an example of a black hole, consider $\mathsf{fix}\, f.f$. As an example of an expression that is *not* a black hole, yet fails to terminate consider $(\mathsf{fix}\, f.\lambda y.f\,(y+1))\,1$. It is instructive to simulate the execution of this expression.

$$\begin{array}{ll} & \langle \cdot, (\mathsf{fix}\, f.\lambda y.f\,(y+1))1\rangle \\ \mapsto & \langle (l = \lambda y.l\,(y+1)), l\,1\rangle \\ \mapsto & \langle (l = \lambda y.l\,(y+1)), (\lambda y.l(y+1))\;1\rangle \\ \mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1), l\,(l_1+1)\rangle \\ \mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1), (\lambda y.l(y+1))\,(l_1+1)\rangle \\ \mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1, l_2 = l_1+1), l\,(l_2+1)\rangle \\ \mapsto & \ldots \end{array}$$

In order to detect black holes and take appropriate action we would allow thunks of the form $l{=}\bullet$ and replace the first rule by

$$\frac{\langle (H_1, l{=}\bullet, H_2), e\rangle \mapsto \langle (H_1', l{=}\bullet, H_2'), e'\rangle}{\langle (H_1, l{=}e, H_2), l\rangle \mapsto \langle (H_1', l{=}e', H_2'), l\rangle}$$

$$\frac{}{\langle (H_1, l{=}\bullet, H_2), l\rangle \mapsto \langle (H_1, l{=}\bullet, H_2), \mathsf{BlackHole}\rangle}$$

where BlackHole is a new error expression that must be propagated to the top level as shown in a previous lecture on run-time exceptions and errors.

Futures. Next we consider *futures*. The idea is that an expression $\mathsf{future}(e)$ spawns a parallel computation of e while returning immediately a pointer to the resulting value. If the resulting value is ever actually needed we say we are *touching* the future. When we touch the future we block until the parallel computation of its value has succeeded. However, in most situations we can pass around the future, construct bigger values, etc.

There are two principal differences to call-by-need as shown above. The first is that a future is treated as a value. This is important because unlike in call-by-need, we are here in a call-by-value setting. Secondly, the computation of the future may proceed asynchronously, instead of being completed in full exactly the first time it is accessed. However, it is similar in the sense that once a future has been computed, its value is available everywhere it is referenced.

The typing rule for futures in source programs is exceedingly simple, since we consider futures related only to how a program executes (sequentially or in parallel), but not what it computes.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{future}(e) : \tau}$$

Process labels l that arise during computation are given types just as stores or heaps are given types. Moreover, labels l are treated as values, which forces us to refine the value inversion lemma if we want to prove the progress theorem.

To describe such a computation we have to describe the overall state of all the computing threads. For this, we just use H, as defined above.

$$\text{Processes} \qquad H ::= \cdot \mid H, l{=}e$$

In this interpretation, labels l are thread identifiers, and $l{=}v$ represents a finished thread. So overall computation proceeds as in

$$H \mapsto H'$$

which non-deterministically selects a process that can proceed (that is, not finished or blocked) and makes a step. The judgment of making a step in the network of parallel processes is

$$\langle H, e\rangle \mapsto \langle H', e'\rangle$$

where H' may contain a new thread spawned by the step of e. Unlike call-by-need evaluation, this judgment cannot change any binding in H; this is reserved for the primary judgment. We start the overall computation of an expression e as a single process $l_0{=}e$ and we are finished when we have reached a state where *all* processes have the form $l{=}v$.

In order to be able to prove a progress theorem, we would like to maintain an order between the processes which reflects possible dependencies.

That is, a process can refer to labels on its left, but not to itself or processes to its right.

The first rule non-deterministically selects a thread to perform a step. In this setting, a process can never refer to itself, because we have no recursive futures. Of course, we may have futures whose computation is recursive.

$$\frac{\langle H_1, e\rangle \mapsto \langle H_1', e'\rangle}{(H_1, l{=}e, H_2) \mapsto (H_1', l{=}e', H_2)} \qquad \frac{}{l \ \mathsf{value}}$$

The rules for the judgment $\langle H, e\rangle \mapsto \langle H', e'\rangle$ are the usual call-by-value rules, threading through H. It is only changed or referenced in the following two rules.

$$\frac{v \ val}{\langle (H_1, l{=}v, H_2), l\rangle \mapsto \langle (H_1, l{=}v, H_2), v\rangle} \qquad \frac{}{\langle H, \mathsf{future}(e)\rangle \mapsto \langle (H, l{=}e), l\rangle}$$

Because l is a value, it can be passed around, or looked up (in case the thread l has finished). This introduces some local non-determinism into expressions such as $\mathtt{apply}(l, e)$ because l could be looked up, or e could be reduced. In the end, the difference is not observable in a call-by-value language without effects. It could also be removed with some additional machinery, but we do not pursue this here, since non-determinism remains anyway due to the selection of the process to step.

Notice that an expression such as $\mathtt{apply}(l, v)$ is blocked until the thread computing l can completed. This is because it not a value, yet cannot be reduced.

The process selection rule must be prescient in this formulation, because we must traverse a thread expression to see if it is finished, can make a step, or is blocked, waiting for another thread to finish. This is a feature generally true for a small-step semantics with search rules. In a semantics with an evaluation stack, this can be avoided because the sub-expression to be evaluated is isolated at the top level of the state.

Note that the left-to-right ordering between processes is necessary to guarantee progress because it prevents a situation where two processes wait for each other to finish. This situation is referred to as a *deadlock*. It is instructive to compute an example of such a process configuration.

The typing judgment on process configurations must take this into account. It has the form $H : \Lambda$, where Λ assigns types to processes. We also generalize the typing judgment for expressions to allow labels to occur—

they are simply propagated except in the one rule shown below.

$$\frac{}{\cdot : \cdot} \qquad \frac{H : \Lambda \quad \Lambda; \cdot \vdash e : \tau}{(H, l{=}e) : (\Lambda, l{:}\tau)}$$

$$\frac{l{:}\tau \text{ in } \Lambda}{\Lambda; \Gamma \vdash l : \tau}$$

The preservation theorem is not difficult to formulate.

Theorem 1 (Preservation)

(i) If $H : \Lambda$ *and* $\Lambda; \cdot \vdash e : \tau$ *and* $\langle H, e\rangle \mapsto \langle H', e'\rangle$ *then there is a* $\Lambda' \supseteq \Lambda$ *such that* $H' : \Lambda'$ *and* $\Lambda'; \cdot \vdash e' : \tau$.

(ii) If $H : \Lambda$ *and* $H \mapsto H'$ *then there is a* $\Lambda' \supseteq \Lambda$ *such that* $H' : \Lambda'$.

Proof: By induction on the derivation of the step relation, applying inversion on the typing assumptions. ■

The progress theorem requires more care. We first formalize the notion of a terminal state.

$$\frac{}{\cdot \text{ terminal}} \qquad \frac{H \text{ terminal} \quad v \text{ value}}{(H, l{=}v) \text{ terminal}}$$

Theorem 2 (Progress)

(i) If $H_1 : \Lambda_1$, H_1 terminal, *and* $\Lambda_1; \cdot \vdash e : \tau$ *then either*

(a) e value, *or*

(b) there exists H_1' *and* e' *such that* $\langle H_1, e\rangle \mapsto \langle H_1', e'\rangle$

(ii) If $H : \Lambda$ *then either*

(a) H terminal, *or*

(b) there exists H' *such that* $H \mapsto H'$

Proof: For (i) by induction on the derivation of $\Lambda_1; \cdot \vdash e : \tau$, using a generalization of value inversion that permits labels. Labels must be defined in H_1 and bound to values (since H_1 is terminal), thereby assuring progress.

For (ii) by appeal to (i) given $H = H_1, l{=}e, H_2$, where e is not a value. Such a decomposition must be possible if H is not terminal. ■

We close this lecture with a two examples of programs written using the future construct. These have been adapted from Halstead's paper, but are present in ML assuming a construct `future(e)`. A simple sequential simulation is simply to define `future` as the identity function.

The first example is the insertion of a node into an ordered binary tree. An ordered binary tree is either `Empty`, a data-carrying `Leaf(x)`, or a node `Node(left,y,right)` where `y` is a discriminator so that every element in the left subtree `left` is smaller or equal to `y`, and every element in the right subtree `right` is larger than `y`.

The parallelism in this example is the possibility to spawn a thread at each recursive call to `insert`, which returns immediately and continues insertion of the subtree. Thereby, if we insert several elements in a row, the computations can ripple down the tree simultaneously almost in a pipeline structure (although there is no assumption that the operations are indeed performed in lock-step).

```
datatype Tree =
    Empty
  | Leaf of int
  | Node of Tree * int * Tree
fun insert (x, Empty) = Leaf(x)
  | insert (x, tree as Leaf(y)) =
    if y < x
      then Node (tree, y, Leaf(x))
    else Node (Leaf(x), x, tree)
  | insert (x, Node(left, y, right)) =
    if y < x
      then Node (left, y, future (insert (x, right)))
    else Node (future (insert (x, left), y, right))
```

As a second example, we consider `quicksort`, implemented on lists. It first partitions a list into elements smaller and greater than a pivot element (the first element in the list) and then sorts the sublists in parallel before appending them. There is also a smaller amount of parallelism in the `partition` function shown below.

```
fun quicksort (nil, acc) = acc
  | quicksort (x::l, acc) =
    let
      val (smaller, greater) = partition (x, l)
    in
      quicksort (smaller,
                 x::future (quicksort (greater, acc)))
    end
and partition (x, nil) = (nil, nil)
  | partition (x, y::l) =
    let
      val parts = future (partition (x, l))
    in
      if y < x
        then (y::future(#1(parts)), future (#2(parts)))
      else (future (#1(parts)), y::future (#2(parts)))
    end
```

It is instructuve to consider the above function without the `future` construct and systematically search for opportunities of parallelism.

Lecture Notes on
The Curry-Howard Isomorphism

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 23
November 18, 2004

In this lecture we explore an interesting connection between logic and programming languages. In brief, logical proofs embody certain constructions which may be interpreted as programs. Under this interpretation, propositions become types. It was first observed by the logicians Haskell Curry [~1960] and William Howard [~1969] in different contexts that this is in fact an isomorphism: in a certain fragment of logic, every proof describes a program and every program describes a proof.

We will make the same observation here, using the methodology of judgments that we have used to far in the course, applied to the development of principles of logical reasoning. This formulation is due to Per Martin-Löf [~1983].

What is the meaning of a proposition? Martin-Löf argues that to understand the meaning of a proposition means to understand when it is true. Consequently, in order to explain the meaning of a logical connective, we have to explain how to derive that a proposition with this connective is true.

Thus the most basic of all judgments of logic is that of truth, written as A true.

As the simplest example of a connective, consider conjunction. We say that $A \wedge B$ true if A true and B true. Written as an inference rule:

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge I$$

We refer to this as an *introduction rule* because it introduces a connective in the conclusion (here '$\wedge$'). The introduction rule tells us how to conclude that $A \wedge B$ is true, thereby defining its meaning.

The next question is how can we use the information that $A \wedge B$ true. According to the above explanation, if we know $A \wedge B$ true, we should know the two premises, that is A true and B true.

$$\frac{A \wedge B \text{ true}}{A \text{ true}} \wedge E_1 \qquad \frac{A \wedge B \text{ true}}{B \text{ true}} \wedge E_2$$

We refer to these as *elimination rules*, because they eliminate a connective in the premise (here '$\wedge$').

If we think of the introduction rule as defining the meaning of the connective, how do we know that the elimination rules that we developed from it are actually correct? We will consider this question later in this lecture by introducing the notions of local soundness and completeness of the rules. For now, we continue by filling out the store of available logical connectives.

The next one we want to consider is implication. When is $A \supset B$ true? From our experience with proofs we know that in order to conclude $A \supset B$ true we assume that A is true and try to prove that B is true. If we want to take this as a definition, we need a hypothetical judgment. We write A_1 true, $\ldots, A_n$ true $\vdash A$ true for such a hypothetical judgment with assumptions A_1 true through A_n true. We abbreviate a collection of hypotheses with H. From the nature of reasoning from hypotheses we obtain for free a hypothesis rule and a substitution principle.

$$\frac{}{H_1, A \text{ true}, H_2 \vdash A \text{ true}}\ \mathit{Hyp}$$

Unfortunately, there is a small ambiguity here: there may be several hypotheses A true and from the form of the rule above we cannot tell which one was meant. In order to make this unambiguous we record the number of the assumption that was used.

Hypothesis rule.

$$\frac{}{A_1 \text{ true}, \ldots, A_i \text{ true}, \ldots, A_n \text{ true} \vdash A_i \text{ true}}\ \mathit{Hyp}^i$$

Substitution principle.

If $H_1 \vdash A$ true and H_1, A true, $H_2 \vdash C$ true then $H_1, H_2 \vdash C$ true.

The latter is called a substitution principle because in order to obtain evidence for C true we substitute derivations of A true for uses of the assumption A true.

Now we have the concepts in place to be able to define implication by its introduction rule.

$$\frac{H, A \text{ true} \vdash B \text{ true}}{H \vdash A \supset B \text{ true}} \supset I$$

The elimination rule is based on the substitution principle. Assume we know that $A \supset B$ true. By the above rule this means B true under the assumption that A true. Now, if we had a proof of A true we could substitute it for uses of the assumption in the proof of B true. As a rule:

$$\frac{H \vdash A \supset B \text{ true} \quad H \vdash A \text{ true}}{H \vdash B \text{ true}} \supset E$$

Now we can prove, for example, that $(A \wedge B) \supset (B \wedge A)$ true.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \text{ true} \vdash A \wedge B \text{ true}}\, Hyp^1}{A \wedge B \text{ true} \vdash B \text{ true}} \wedge E_2 \quad \dfrac{\dfrac{}{A \wedge B \text{ true} \vdash A \wedge B \text{ true}}\, Hyp^1}{A \wedge B \text{ true} \vdash A \text{ true}} \wedge E_1}{A \wedge B \text{ true} \vdash B \wedge A \text{ true}} \wedge I}{\cdot \vdash (A \wedge B) \supset (B \wedge A) \text{ true}} \supset I$$

Note that this is holds for any propositions A and B, that is, it is a schematic derivation just like inference rules are schematic.

We continue our analysis of logical connectives with disjunction $A \vee B$. The disjunction is true if either of the disjuncts is true. This means we have two introduction rules.

$$\frac{H \vdash A \text{ true}}{H \vdash A \vee B \text{ true}} \vee I_1 \qquad \frac{H \vdash B \text{ true}}{H \vdash A \vee B \text{ true}} \vee I_2$$

In order to determine the elimination rule, we must consider how to use the knowledge that $A \vee B$ true. Clearly, we do not know which of A true or B true holds. This means if we are trying to prove C true and we know $A \vee B$ true, we must be able to show C true no matter whether A true or B true. In other words, we must proceed with a proof by cases. In the form of an elimination rule:

$$\frac{H \vdash A \vee B \text{ true} \quad H, A \text{ true} \vdash C \text{ true} \quad H, B \text{ true} \vdash C \text{ true}}{H \vdash C \text{ true}} \vee E$$

As a sample proof, consider the statement

If A or B implies C, then A implies C.

Formally:

$$((A \vee B) \supset C) \supset (A \supset C)\ \mathsf{true}$$

The proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A \vee B) \supset C\ \mathsf{true}, A\ \mathsf{true} \vdash A\ \mathsf{true}}\ Hyp^2}{(A \vee B) \supset C\ \mathsf{true}, A\ \mathsf{true} \vdash A \vee B\ \mathsf{true}}\ \vee I_1 \quad \dfrac{}{(A \vee B) \supset C\ \mathsf{true}, A\ \mathsf{true} \vdash (A \vee B) \supset C\ \mathsf{true}}\ Hyp^1}{(A \vee B) \supset C\ \mathsf{true}, A\ \mathsf{true} \vdash C\ \mathsf{true}}\ \supset E}{(A \vee B) \supset C\ \mathsf{true} \vdash A \supset C\ \mathsf{true}}\ \supset I}{\cdot \vdash ((A \vee B) \supset C) \supset (A \supset C)\ \mathsf{true}}\ \supset I$$

Next we look at some degenerate cases. Consider truth ($\top$) as a logical constant. It should be provable no matter what assumptions we have.

$$\dfrac{}{H \vdash \top\ \ \mathsf{true}}\ \top I$$

Because we put no information into the proof of $\top$, we can obtain no information out. Therefore, there is no elimination rule for $\top$. We can observe that $\top$ is like a 0-ary version of conjunction: $\wedge I$ has two premises and consequently we have two elimination rules ($\wedge E_1$ and $\wedge E_2$), while $\top I$ has no premises and consequently no elimination rules.

Now consider falsehood ($\bot$). It represents a contradiction and should therefore not be provable. In other words, there is no introduction rule. Conversely, if we know $\bot$ true we should be able to conclude anything.

$$\dfrac{H \vdash \bot\ \ \mathsf{true}}{H \vdash C\ \ \mathsf{true}}\ \bot E$$

We can recognize falsehood as a disjunction of zero alternatives. Whereas there are two introduction rules for $\vee$ and therefore two cases to consider in the elimination rule, there are no introduction rules for $\bot$ and therefore no branches in the elimination rule.

Figure 1 summarizes the rules, adding hypotheses to the first rules about conjunction in the straightforward way. We list the introduction rules in the left column and elimination rules in the right column. We have stacked the premises of the $\vee E$ rules purely for typographical reasons.

A natural question is if these are all the logical connectives we may be interested in, and if the given rules define logical reasoning completely if restricted to the considered connectives. If we ignore universal and existential quantification, then the main missing connectives are logical equiv-

$$\frac{}{A_1\ \mathsf{true}, \ldots, A_i\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A_i\ \mathsf{true}}\ \mathit{Hyp}^i$$

$$\frac{H \vdash A\ \mathsf{true} \quad H \vdash B\ \mathsf{true}}{H \vdash A \wedge B\ \mathsf{true}}\ \wedge I \qquad \frac{H \vdash A \wedge B\ \mathsf{true}}{H \vdash A\ \mathsf{true}}\ \wedge E_1 \qquad \frac{H \vdash A \wedge B\ \mathsf{true}}{H \vdash B\ \mathsf{true}}\ \wedge E_2$$

$$\frac{}{H \vdash \top\ \mathsf{true}}\ \top I \qquad \text{no } \top E \text{ rule}$$

$$\frac{H, A\ \mathsf{true} \vdash B\ \mathsf{true}}{H \vdash A \supset B\ \mathsf{true}}\ \supset I \qquad \frac{H \vdash A \supset B\ \mathsf{true} \quad H \vdash A\ \mathsf{true}}{H \vdash B\ \mathsf{true}}\ \supset E$$

$$\frac{H \vdash A\ \mathsf{true}}{H \vdash A \vee B\ \mathsf{true}}\ \vee I_1 \qquad \frac{H \vdash B\ \mathsf{true}}{H \vdash A \vee B\ \mathsf{true}}\ \vee I_2 \qquad \frac{\begin{array}{l} H \vdash A \vee B\ \mathsf{true} \\ H, A\ \mathsf{true} \vdash C\ \mathsf{true} \\ H, B\ \mathsf{true} \vdash C\ \mathsf{true} \end{array}}{H \vdash C\ \mathsf{true}}\ \vee E$$

$$\text{no } \bot I \text{ rule} \qquad \frac{H \vdash \bot\ \mathsf{true}}{H \vdash C\ \mathsf{true}}\ \bot E$$

Figure 1: Rules of Intuitionistic Propositional Logic

alence $A \equiv B$ and negation $\neg A$. These can be easily considered abbreviations, using

$$\begin{array}{rcl} A \equiv B & = & (A \supset B) \wedge (B \supset A) \\ \neg A & = & A \supset \bot \end{array}$$

On the question of the completeness of these rules, a debate is possible. With the right proof-theoretic analysis we can show that $A \vee \neg A$ true is *not* provable for an arbitrary A in this logic, essentially because we can prove neither A true nor $\neg A$ true, which are the two possibilities if we consider the introduction rules for disjunction ($\vee$).

The logic we have developed so far is, for historical reasons, called *intuitionistic logic*. If we also allow arbitrary instances of the axiom schema of *excluded middle* (*XM*),

$$\frac{}{H \vdash A \vee \neg A\ \mathsf{true}}\ \mathit{XM}$$

we obtain what is called *classical logic*. Note how classical logic, in a rule rather difficult to motivate, destroys the design principles and attempts at explaining the meaning of the connectives. For example, to understand the meaning of disjunction it is no longer sufficient to understand its introduction rules, but we must also understand the law of excluded middle, which contains and appeal to negation and falsehood. All is not lost, but we can say that the Curry-Howard isomorphism (the subject of this lecture whose explanation is yet to come) will fail in the presence of the law of excluded middle.

For the rest of this lecture we will only be interested in the intuitionistic logic as defined with the rules in Figure 1. The first observation is that a derivable judgment $H \vdash A\ \mathsf{true}$ does not contain any information about its derivation. When we assert $H \vdash A\ \mathsf{true}$ is derivable we mean that it has a derivation, but we do not exhibit such a derivation. This makes it difficult to convince someone else of the truth of A under assumptions H. So what we would like to do is to enrich the judgment with a *proof term* M which contains enough information to reconstruct the derivation.

So we would like to uniformly replace the judgment $A\ \mathsf{true}$ with $M : A$ (read: *M is a proof of A*). For assumptions $A\ \mathsf{true}$ we do not actually have a proof of A, we *assume* that there is a proof. We model this by using a *variable*, where each assumption is labeled by a distinct variable.

So we want to translate a judgment

$$A_1\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A\ \mathsf{true}$$

to the form

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : A$$

in such a why that the derivation of the first judgment can be reconstructed directly from M. First, the hypothesis rule is straightforward:

$$\frac{}{A_1\ \mathsf{true}, \ldots, A_i\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A_i\ \mathsf{true}}\ \mathit{Hyp}^i$$

becomes

$$\frac{}{x_1{:}A_1, \ldots, x_i{:}A_i, \ldots, x_n{:}A_n \vdash x_i : A_i}\ \mathit{Hyp}$$

Since the assumptions are labeled by distinct variables, we no longer need to annotate the justification with an integer and we just write *Hyp*.

The remaining rules mention a collection of hypotheses $H = (A_1\ \mathsf{true}, \ldots, A_n\ \mathsf{true})$ which we annotate uniformly with distinct variables, leading to a context $\Gamma = (x_1{:}A_1, \ldots, x_n{:}A_n)$.

We begin the logical connectives with conjunction. A proof of a conjunction $A \wedge B$ by the introduction rule $\wedge I$ consists of a pair of proofs, one of A and one for B.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{pair}(M, N) : A \wedge B}\ \wedge I$$

We can recover the old rule by ignoring the proof terms, which immediately shows that the rule is sound with respect to the truth judgment. The two elimination rules can be considered as extracting components of this pair of proofs, which is why we use the suggesting names fst and snd.

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{fst}(M) : A}\ \wedge E_1 \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathsf{snd}(M) : B}\ \wedge E_2$$

The main observations of the Curry-Howard isomorphism should now already be visible:

1. (**Propositions-as-types**) Propositions of logic correspond to types of a programming language
2. (**Proofs-as-programs**) Proofs in logic correspond to expressions in a programming language
3. (**Proof-checking-as-type-checking**) Verifying the correctness of a proof corresponds to type-checking its corresponding expression.

We will consider later how computations are interpreted.

Now we go back to the logical connectives, considering implication. We need to account for the fact that the introduction rule ($\supset I$) introduces a new hypotheses. In the proof term this is handled as a binding construct. To make the reconstruction problem unambiguous we record A in the expression. A more standard notation for $\mathsf{fn}(A, x.M)$ would be $\lambda x{:}A.M$.

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \mathsf{fn}(A, x.M) : A \supset B}\ \supset I$$

In words: a proof of $A \supset B$ is a function which maps a proof of A to a proof of B. This is the functional interpretation of implication in intuitionistic logic. The elimination rule just applies such a function to an argument proof term.

$$\frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{apply}(M, N) : B}\ \supset E$$

In summary, logical implications corresponds to function types, analogous to the way that logical conjunctions correspond to product types.

It is not hard to guess that logical disjunction will correspond to disjoint sum types.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}(B, M) : A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr}(A, M) : A \vee B} \vee I_2$$

Proof by cases corresponds to the case construct over disjoint sums.

$$\frac{\Gamma \vdash M : A \vee B \quad \Gamma, x{:}A \vdash N : C \quad \Gamma, y{:}B \vdash P : C}{\Gamma \vdash \mathsf{case}(M, x.N, y.P) : C} \vee E$$

The logical constant $\top$ just becomes the unit type, and the logical constant $\bot$ the void (empty) type.

$$\frac{}{\Gamma \vdash \mathsf{unit} : \top} \top I$$

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathsf{abort}(C, M) : C} \bot E$$

The following table summarizes the correspondence between propositions and types under the Curry-Howard isomorphism:

Conjunction	$A \wedge B$	$\tau \times \sigma$	Product Type
Truth	$\top$	1	Unit Type
Implication	$A \supset B$	$\tau \to \sigma$	Function Type
Disjunction	$A \vee B$	$\tau + \sigma$	Sum Type
Falsehood	$\bot$	0	Void Type

As example, consider the following sample proofs, now written out in proof terms rather than full derivations. We have elided some (in this case, redundant) types[1] in the terms. As remarked in the lecture about bi-directional type-checking, if the term is *normal* and the type is known on the outside, then no internal type annotations are necessary.

$$\begin{array}{rcl} \mathsf{fn}(x.\mathsf{pair}(\mathsf{snd}(x), \mathsf{fst}(x))) & : & (A \wedge B) \supset (B \wedge A) \\ \mathsf{fn}(x.\mathsf{fn}(y.\mathsf{apply}(x, \mathsf{inl}(y)))) & : & ((A \vee B) \supset C) \supset (A \supset C) \end{array}$$

At this point it should be easy to see that we could actually let ML do some of the proof-checking for us. For example, with definitions

[1] that is, propositions

```
fun pair(x,y) = (x,y);
fun fst(x,y) = x;
fun snd(x,y) = y;
```

the first proof term above can be written as

```
- (fn x => pair (snd x, fst x));
val it = fn : 'a * 'b -> 'b * 'a
```

which constitutes a proof of $(A \wedge B) \supset (B \wedge A)$. The principal difficulty is that the presence of effects and recursion destroys the isomorphism. For example,

```
fun loop(x) = loop(x);
val loop = fn : 'a -> 'b
```

but the corresponding proposition, $A \supset B$ cannot be true in general. This means type-checking along in ML does not implement proof-checking; we also have to verify (by hand) the absence of effects and recursion.

We will not formalize this here, but it follows by straightforward inductions that for a derivation $\mathcal{D}$ of $A_1\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A\ \mathsf{true}$ we can systematically construct a derivation of $x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : A$. Moreover, if $x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : A$ then by erasure of terms (and appropriate labeling of the hypothesis rule) we can construct a derivation of $A_1\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A\ \mathsf{true}$. These two translations are inverses of each other. In other words, the correspondence is really an ismorphism between proofs and programs.

It remains to consider what the role of *computation* is on the logical side. The whole construction seems too beautiful and elegant for the operational semantics of programs to be a simple accident without logical counterpart. In order to investigate this we return to the question of how to ascertain the "correctness" of the introduction and elimination rules for each connective. For example, it would clearly be unsound to have an elimination rule that allows us to infer $A\ \mathsf{true}$ from $A \vee B\ \mathsf{true}$. But how can we formally reject such an incorrect elimination rule?

In our context here we break down the correctness of the elimination rules with respect to the introduction rules into two questions: *local soundness* and *local completeness*.

Local soundness means that the elimination rules are not too strong. We have to verify that we cannot obtain more knowledge from a judgment by an elimination rule than we put into it by an introduction rule. More formally, we must show that if we introduce a connective and then eliminate it, we could derive the conclusion without this detour.

In the example of conjunction, this property is quite easy to see. We consider the possible combinations of introductions followed by eliminations, of which there are two.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ A\ \mathsf{true} & B\ \mathsf{true}\end{array}}{A \wedge B\ \mathsf{true}}\,\wedge I}{A\ \mathsf{true}}\,\wedge E_1 \qquad \dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ A\ \mathsf{true} & B\ \mathsf{true}\end{array}}{A \wedge B\ \mathsf{true}}\,\wedge I}{B\ \mathsf{true}}\,\wedge E_2$$

In the first case, we can eliminate the detour because $\mathcal{D}$ is already a derivation of the conclusion, in the second case it is $\mathcal{E}$. We write this as *local reductions* on proofs that witness the local soundness of the rules.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ A\ \mathsf{true} & B\ \mathsf{true}\end{array}}{A \wedge B\ \mathsf{true}}\,\wedge I}{A\ \mathsf{true}}\,\wedge E_1 \quad \mapsto \quad \begin{array}{c}\mathcal{D} \\ A\ \mathsf{true}\end{array}$$

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ A\ \mathsf{true} & B\ \mathsf{true}\end{array}}{A \wedge B\ \mathsf{true}}\,\wedge I}{B\ \mathsf{true}}\,\wedge E_2 \quad \mapsto \quad \begin{array}{c}\mathcal{E} \\ B\ \mathsf{true}\end{array}$$

If we annotate the derivations with proof terms we see that each local reduction is a *rule of computation* on proof terms.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ M : A & N : B\end{array}}{\mathsf{pair}(M,N) : A \wedge B}\,\wedge I}{\mathsf{fst}(\mathsf{pair}(M,N)) : A}\,\wedge E_1 \quad \mapsto \quad \begin{array}{c}\mathcal{D} \\ M : A\end{array}$$

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ M : A & N : B\end{array}}{\mathsf{pair}(M,N) : A \wedge B}\,\wedge I}{\mathsf{snd}(\mathsf{pair}(M,N)) : B}\,\wedge E_2 \quad \mapsto \quad \begin{array}{c}\mathcal{E} \\ N : B\end{array}$$

Written out using only proof terms:

$$\begin{array}{rcl} \mathsf{fst}(\mathsf{pair}(M, N)) & \mapsto & M \\ \mathsf{snd}(\mathsf{pair}(M, N)) & \mapsto & N \end{array}$$

So proof reduction arises from showing that a "detour", that is, the introduction of a connective immediately followed by its elimination, can be avoided, leading to a "more direct" proof of the conclusion. The logically important property of this proof reduction is that it witnesses local soundness: we cannot get more information out of the truth of a proposition than we put into it.

Under the Curry-Howard isomorphism, computation then arises from a notion of proof reduction by imposing a particular strategy of reduction. For ML, this strategy is characterized by the search rules that specify where a reduction may take place. It seems that nothing about the logical meaning of a program forces the particular strategy adopted by ML, which means that the logical reading underdetermines how to evaluate programs but instead provides only the basic building blocks, namely the reductions.

To extend our analysis of proof reductions to implications, we need to consider substitution. Recall the substitution principle:

> If $H_1 \vdash A$ true and H_1, A true, $H_2 \vdash C$ true then $H_1, H_2 \vdash C$ true.

If we annotate this with proof terms we obtain:

> If $\Gamma_1 \vdash M : A$ and $\Gamma_1, x{:}A, \Gamma_2 \vdash N : C$
> then $\Gamma_1, \Gamma_2 \vdash \{M/x\}N : C$.

Now the pure proof reduction for an implication introduction followed by its elimination has the form

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ H, A \text{ true} \vdash B \text{ true}\end{array}}{H \vdash A \supset B \text{ true}} \supset I \qquad \begin{array}{c}\mathcal{E}\\ H \vdash A \text{ true}\end{array}}{H \vdash B \text{ true}} \supset E \qquad \mapsto \qquad \begin{array}{c}\mathcal{D}'\\ H \vdash B \text{ true}\end{array}$$

where the existence of $\mathcal{D}'$ is justified by the substitution property applied to $\mathcal{E}$ and $\mathcal{D}$. With proof terms:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \mathsf{fn}(A, x.M) : A \supset B} \supset I \qquad \begin{array}{c}\mathcal{E}\\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash \mathsf{apply}(\mathsf{fn}(A, x.M), N) : B} \supset E \qquad \mapsto \qquad \begin{array}{c}\mathcal{D}'\\ \Gamma \vdash \{N/x\}M : B\end{array}$$

Written out using only proof terms:

$$\mathsf{apply}(\mathsf{fn}(A, x.M), N) \quad \mapsto \quad \{N/x\}M$$

For sums, we have to check two combinations of introduction followed by an elimination, because there are to rules for disjunction introduction. We leave it to the reader to write out the proof reduction that witnesses local soundness. The corresponding proof term reductions are

$$\begin{array}{lcl} \mathsf{case}(\mathsf{inl}(B, M), x.N, y.P) & \mapsto & \{M/x\}N \\ \mathsf{case}(\mathsf{inr}(A, M), x.N, y.P) & \mapsto & \{M/y\}P \end{array}$$

For truth ($\top$) and falsehood ($\bot$) no local reductions arise, because truth has only and introduction and falsehood only an elimination. Consequently, there are no reduction rules for the unit and void types, which is consistent with our definition of MinML.

One further remark regarding the connection between proof reductions and rules of computation. The fact that proof reductions transform one valid proof of $\Gamma \vdash M : A$ to another valid proof $\Gamma \vdash M' : A$ ensures *type preservation* for the corresponding computation rules in the programming languages.

There is a second check that is usually applied to the introduction and elimination rules for a connective to verify that the elimination rules are strong enough to recover all the information that has been put into a proposition. We have to verify that if we assume we have a proof of a proposition, we can apply elimination rules in such a way that we can reconstruct a proof of the original proposition by an introduction rule. We call this property *local completeness*, which is witnessed by a local expansion. However, local expansions do not have an immediate computational meaning, but are connected to the canonical forms property (also called value inversion). We do not explore this connection further and just show an example.

$$\overset{\mathcal{D}}{H \vdash A \wedge B\ \mathsf{true}} \quad \Longrightarrow \quad \dfrac{\dfrac{\overset{\mathcal{D}}{H \vdash A \wedge B\ \mathsf{true}}}{H \vdash A\ \mathsf{true}}\wedge E_1 \quad \dfrac{\overset{\mathcal{D}}{H \vdash A \wedge B\ \mathsf{true}}}{H \vdash B\ \mathsf{true}}\wedge E_2}{H \vdash A \wedge B\ \mathsf{true}}\wedge I$$

With proof terms:

$$\overset{\mathcal{D}}{\Gamma \vdash M : A \wedge B} \quad \Longrightarrow \quad \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma \vdash M : A \wedge B}}{\Gamma \vdash \mathsf{fst}(M) : A}\wedge E_1 \quad \dfrac{\overset{\mathcal{D}}{\Gamma \vdash M : A \wedge B}}{\Gamma \vdash \mathsf{snd}(M) : B}\wedge E_2}{\Gamma \vdash \mathsf{pair}(\mathsf{fst}(M), \mathsf{snd}(N)) : A \wedge B}\wedge I$$

Or purely on terms (indicating the type of the left-hand side)

$$M : A \wedge B \implies \mathsf{pair}(\mathsf{fst}(M), \mathsf{snd}(M))$$

Lecture Notes on Program Equivalence

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 24
November 30, 2004

When are two programs equal? Without much reflection one might say that two programs are equal if they evaluate to the same value, or if both of them run forever. This explicitly ignores the issue of effects, and we will continue to think about a pure language until later in this lecture. So, in a pure language the statement above reduces the equality of programs to the equality of values. But when should two values be equal? For example, how about the following two functions.

$$\begin{array}{lcl} id_1 & = & \lambda x.x \\ id_2 & = & \lambda x.x + 0 \end{array}$$

The first observation is that they are both values, so they definitely will not diverge.

Now, id_1 and id_2 return the same integer when applied to an integer, but id_1 has more types than id_2. We conclude from that the we should compare two values at a type. In general, the judgment has the form

$$v \simeq v' : \tau$$

where we assume that $\cdot \vdash v : \tau$ and $\cdot \vdash v' : \tau$. Here we want to ask if

$$(\lambda x.x) \simeq (\lambda x.x + 0) : \mathsf{int} \to \mathsf{int?}$$

The answer to this question depends on our point of view. If we care about efficiency, for example, they are not equal since the left-hand side always takes one fewer step than the right-hand side. If we care about the syntactic form of the function, they are not equal either. On the other hand,

if we only care about the result of the function when applied to all possible arguments, then the two should be considered equal at the given type, since both of them are (mathematically) the identity function on integers.

In this lecture we are concerned with *observational equivalence* between programs: we consider to programs (and values) equal if whatever we can observe about their behavior is identical. In pure functional languages, the only thing you can observe about a program is the final value it returns. But there are further restrictions. For example, we cannot observe the internal structure of functions. In implementations, they have been compiled to machine code—all we see is a token such as `fn` indicating the given value cannot be printed.

If we cannot observe the structure of a function, what can be observe about a function? We can apply it to arguments and observe its result. But this result may again be a function whose structure we cannot see directly. It appears we are moving in a vicious circle, trying to define observational equivalence of functions in terms of itself.

Fortunately, there is a way out. We once again use *types* in order to create order out of chaos. In our example above, the functions $\lambda x.x$ and $\lambda x.x + 0$ should be equal at type $\mathsf{int} \to \mathsf{int}$ because apply both of them to equal arguments of type int will always yield equal results of type int. And values of type int are directly observable—they form a basic data type of our language.

Using this intuition we can now define two relations of observational equivalence for a pure, call-by-value language by simultaneous induction on the structure of a type of the expressions we are comparing. We write $e \Uparrow$ if the evaluation of e does not terminate. We also use the convention that when we write $e \cong e' : \tau$ that $\cdot \vdash e : \tau$ and $\cdot \vdash e' : \tau$ and similarly for values without restating this every time.

$$
\begin{array}{lll}
e \cong e' : \tau & \text{iff} & \text{either } e \Uparrow \text{ and } e' \Uparrow \\
 & & \text{or } e \mapsto^* v \text{ and } e' \mapsto^* v' \text{ with } v \simeq v' : \tau \\
 & & \\
v \simeq v' : \mathsf{int} & \text{iff} & v = v' = n \text{ for an integer } n. \\
v \simeq v' : \mathsf{bool} & \text{iff} & v = v' = \mathsf{true} \text{ or } v = v' = \mathsf{false} \\
v \simeq v' : \tau_1 \to \tau_2 & \text{iff} & \text{for all } v_1 \simeq v_1' : \tau_1 \text{ we have } v\, v_1 \cong v'\, v_1' : \tau_2
\end{array}
$$

The last clause requires careful analysis. Functions are not observable directly, although we can apply them to arguments to observe their result. The case of values of function type can therefore be summarized as: "*Two functions are equal at type $\tau_1 \to \tau_2$ if they deliver equal results of type τ_2 when applied to equal arguments of type τ_1.*" Note that on the right-hand side the types are smaller than on the left-hand side, so the definition is well-founded. It

is also allowed that neither of the two functions terminates when given equal arguments. This follows from comparing the expressions $v\, v_1$ and $v'\, v_1'$ which have to be evaluated first.

We can use this definition to prove our original assertion that $\lambda x.x \simeq \lambda x.x + 0 : \mathsf{int} \to \mathsf{int}$.

$v_1 \simeq v_1' : \mathsf{int}$	Assumption
$v_1 = v_1' = n$ for some integer n	By definition of $\simeq$
$n \simeq n : \mathsf{int}$	By definition of $\simeq$
$(\lambda x.x)\, n \mapsto^* n$	By definition of $\mapsto$
$(\lambda x.x + 0)\, n \mapsto^* n$	By definition of $\mapsto$
$(\lambda x.x)\, n \cong (\lambda x.x + 0)\, n : \mathsf{int}$	By definition of $\cong$
$(\lambda x.x)\, v_1 \cong (\lambda x.x + 0)\, v_1' : \mathsf{int}$	Since $v_1 = v_1' = n$
$(\lambda x.x) \simeq (\lambda x.x + 0) : \mathsf{int} \to \mathsf{int}$	By definition of $\simeq$

In many cases equivalence proofs are not that straightforward, but require considerable effort. As a slightly more complicated example consider

$$\begin{array}{lcl} id_1 & = & \lambda x.x \\ id_3 & = & \mathsf{rec}\ f.\ \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1) + 1 \end{array}$$

We notice that id_1 and id_3 are in fact *not* equal at type $\mathsf{int} \to \mathsf{int}$ because $id_3\,(-1)$ diverges, while $id_1\,(-1) \mapsto^* -1$. However, when applied to natural numbers, that is, integers greater or equal to 0, then they are observationally equal (both return the argument). In order to capture this we introduce $\mathsf{nat} \leq \mathsf{int}$ under the subset interpretation of subtyping and extend observational equivalence with the clause

$v \simeq v' : \mathsf{nat}$ iff $v = v' = k$ for some $k \geq 0$.

With these definitions we need a lemma, which can be proven by induction on k. For this, we introduce the definition

$$id_3' \quad = \quad \lambda x.\ \mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ id_3\ (x-1) + 1$$

which has the property that $id_3 \mapsto id_3'$ and id_3' is a value. Now we can prove:

For any $k \geq 0$, we have $id_1\, k \cong id_3'\, k : \mathsf{nat}$.

Proof: By induction on k.

Case: $k = 0$. Then $id_1\, 0 \mapsto 0$ and $id_3'\, 0 \mapsto \mathsf{if}\ 0 = 0\ \mathsf{then}\ 0\ \mathsf{else}\ id_3(k-1) + 1 \mapsto^* 0$.

Case: $k = k'+1$. Then $id_1\, k \mapsto k$ and $id_3'\, k \mapsto^* id_3(k-1)+1 \mapsto^* id_3'(k')+1$. By induction hypothesis, $id_3'(k') \cong id_1(k')$ so $id_3'(k') \cong k'$ and $id_3'\, k \mapsto^* k'+1 = k$, which is what we needed to show. ■

From this it follows directly by definition of $\simeq$ that $id_1 = id_3$, since $v_1 \simeq v_1' : \mathsf{nat}$ iff $v_1 = v_1' = k$ for some k and $id_3 \mapsto id_3'$.

Some care must be taken in general to define observational equivalence correctly with respect to what is observable. For example, in a call-by-name language we would have to apply functions to arbitrary expressions, instead of testing them just on values.

It should also be clear that in the presence of effects, be it store effects or control effects, the definition of observational equivalence must be changed substantially to account for the effects.

In the remainder of this lecture we briefly explore the question of equivalence in a setting where we have only effects. In particular, we are no longer interested in termination or the value produced by a computation, but just the externally observable effects it has. This is a fundamental shift in perspective on the notion of computation, but one that is appropriate in the realm of concurrency. For example, we may have server process that never finishes, but forever answers request. It does not return a value (because it never does return), but it interacts with the outside world by receiving requests and sending replies. In this setting, observational equivalence implies that the server answers with equal reply given equal requests. This is a bit imprecise in the setting where we also have non-determinism, that is, a process might evolve in different ways.

For this, we introduce the notion of a *sequential process expression*. Sequential processes can evolve non-deterministically and have externally observable actions, but they do not yet integrate concurrency which is reserved for the next lecture. We start with (observable) actions α which, at present consist either of names a (eventually denoting an input action) and co-names $\bar{a}$ (eventually denoting an output action). A sequential process expression P is defined by the following grammar.

$$P \quad ::= \quad A \mid \alpha_1.P_1 + \cdots + \alpha_n.P_n$$

We write 0 for a sum of zero elements; it corresponds to a process that has terminated (it can take no further actions). Note that "." is not related to variable binding here, it simply separates the prefix α from the process expression P. The *process identifiers* A are defined by, possibly recursive equations

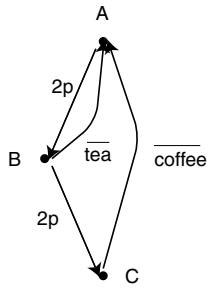
$$A \stackrel{\text{def}}{=} P_A.$$

Sequential process expression evolve in a rather straightforward way. We can unfold a definition of a process identifier, or we can select one non-deterministically from a sum. When such an action is taken, the result is observable. We define a single-step judgment $P \xrightarrow{\alpha} P'$ meaning that P transitions in one step to P' exhibiting action α.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum} \qquad \frac{(A \stackrel{\text{def}}{=} P_A) \quad P_A \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ \text{Def}$$

The Sum rule non-deterministically selects an element of a sum and exhibits action α. Because of the syntax of the language, we cannot replace a part of the sum. We write M and N for sums.

An examples, consider a tea and coffee vending machine with the following informal behavior: if we put in twopence[1] we can obtain tea by pusing an appropriately labeled button, or we can deposit 2 more pennies and the obtain coffee. This machine can be depicted as



and described as a sequential process as follows:

$$A \stackrel{\text{def}}{=} 2\text{p}.(\overline{\text{tea}}.A + 2\text{p}.\overline{\text{coffee}}.A)$$

The vending machine has three states: an initial state A (in which it only waits for the input of 2p), a state B where we can either get the $\overline{\text{tea}}$, or put in another 2p, and a state C where can only ge the $\overline{\text{coffee}}$. We can make this explicit with this alternative definition

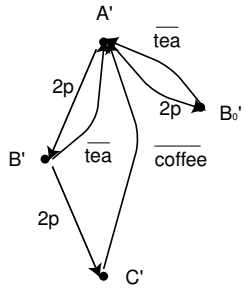
$$\begin{array}{lcl} A & \stackrel{\text{def}}{=} & 2\text{p}.B \\ B & \stackrel{\text{def}}{=} & \overline{\text{tea}}.A + 2\text{p}.C \\ C & \stackrel{\text{def}}{=} & \overline{\text{coffee}}.A \end{array}$$

[1]This example is taken from Robin Milner's book on *Communicating and Mobile Processes: the π-Calculus*, Cambridge University Press, 1999.

Now we return to the question of observational equivalence. If we think just about the actions that the vending machine can exhibit, they can be described by the regular expression:

$$(2\mathrm{p} \cdot (\overline{\mathrm{tea}} + 2\mathrm{p} \cdot \overline{\mathrm{coffee}}))^*.$$

However, this regular expression does not characterize the vending machine as it interacts with its environment. In order to see that, consider the following (broken) vending machine.



$$\begin{array}{lcl} A' & \stackrel{\mathrm{def}}{=} & 2\mathrm{p}.B' + 2\mathrm{p}.B'_0 \\ B' & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{tea}}.A' + 2\mathrm{p}.C' \\ B'_0 & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{tea}}.A' \\ C' & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{coffee}}.A' \end{array}$$

In words, this machine differs from the first one as follows: when we supply it with $2\mathrm{p}$ when in state A', it will non-deterministically go to state B' as before, or go into a new state B'_0 in which we can only obtain $\overline{\mathrm{tea}}$, but not deposit any additional money. Clearly, this machine is broken. However, the sequence of actions it can produce, namely

$$(2\mathrm{p} \cdot (\overline{\mathrm{tea}} + 2\mathrm{p} \cdot \overline{\mathrm{coffee}}) + 2\mathrm{p} \cdot \overline{\mathrm{tea}})^*$$

is exactly the same as for the first machine.

What has gone wrong is the the *reactive* behavior of the system has changed. But this is what we will be interested in when analyzing communicating processes. Here, every input or output will be seen as an interaction with the environment, and then the two vending machines are clearly not equivalent.

In order to capture in what sense they are equivalent we define the notion of *strong simulation*. Let $\mathcal{S}$ be a relation on the states of a process or

between several processes. We say that $\mathcal{S}$ is a *strong simulation* if whenever $P \xrightarrow{\alpha} P'$ and $P \mathcal{S} Q$ then there exists a state Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$. We say that Q strongly simulates P is there exists a strong simulation $\mathcal{S}$ such that $P \mathcal{S} Q$.

For example, the first machine above strongly simulates the second in the sense that there is a strong simulation $\mathcal{S}$ such that $A' \mathcal{S} A$. We write this simulation as $\leq_1$. It is defined by

$$\begin{array}{ll} A' \leq_1 A & \\ B' \leq_1 B & B'_0 \leq_1 B \\ C' \leq_1 C & \end{array}$$

In order to prove that this is a strong simulation we have to verify the conditions in the definition for every transition of the second machine.

Case: $A' \xrightarrow{\text{2p}} B'$ and $A' \leq_1 A$. We have to show there is state Q such that $A \xrightarrow{\text{2p}} Q$ and $B' \leq Q$. $Q = B$ satisfies this condition. We abbreviate this argument in the following case by just showing the relevant transition.

Case: $A' \xrightarrow{\text{2p}} B'_0$ and $A' \leq_1 A$. Then $A \xrightarrow{\text{2p}} B$ and $B'_0 \leq_1 B$.

Case: $B' \xrightarrow{\overline{\text{tea}}} A'$ and $B' \leq_1 B$. Then $B \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

Case: $B' \xrightarrow{\text{2p}} C'$ and $B' \leq_1 B$. Then $B \xrightarrow{\text{2p}} C$ and $C' \leq_1 C$.

Case: $B'_0 \xrightarrow{\overline{\text{tea}}} A'$ and $B'_0 \leq_1 B$. Then $B \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

Case: $C' \xrightarrow{\overline{\text{coffee}}} A'$ and $C' \leq_1 C$. Then $C \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

This covers all cases, so A strongly simulates A'. The perhaps surprising fact is that A' also strongly simulates A, although we need a different relation. We define

$$\begin{array}{l} A \leq_2 A' \\ B \leq_2 B' \\ C \leq_2 C' \end{array}$$

so that B'_0 is not related to any other state. Then $\leq_2$ shows that A' strongly simulates A. Intuitively, this is the case, because the second machine can

simulate every step the first machine can take. It can also exhibit some additional undesired behavior, but this does not matter when we construct a strong simulation.

Now it seems like we have defeated our original purpose, since the two vending machines should not be observationally equivalent, but each one can strongly simulate the other. It turns out that the notion we are interested in is not mutual strong simulation, but *strong bisimulation* which means that there is a *single* relation between the states that acts as a strong simulation in both directions. Under this definition, the two vending machines are not equivalent, because any bi-simulation would have to relate B' and B'_0 to B, but B'_0 could never simulate B because it cannot simulate the transition to C.

In summary, we have isolated the notion of strong bisimulation that we can use to compare the behavior of sequential processes with observable actions and non-deterministic choice. In the next lecture we will make our language of processes richer, allowing for concurrency and interaction.

Lecture Notes on Concurrent Processes

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 25
December 2, 2004

We have seen in the last lecture that by investigating the reactive behavior of systems, we obtain a very different view of computation. Instead of termination and the values of expressions, it is the interactions with the outside world that are of interest. As an example, we showed an important notion of program equivalence, namely strong bisimulation and contrasted it with observational equivalence of computation with respect to values.

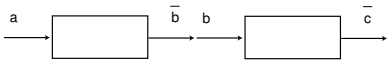
The processes we have considered so far were non-deterministic, but sequential. In this lecture we generalize this to allow for concurrency and also name restriction to obtain a form of abstraction.

In order to model concurrency we allow *process composition*, $P_1 \mid P_2$. Intuitively, this means that processes P_1 and P_2 execute concurrently. Such concurrent processes can interact in a synchronous fashion when one process wants to perform an input action and another process wants to perform a matching output action. As a very simple example, consider two processes A and B plugged together in the following way. A performs input action a and then wants to perform output action $\overline{b}$, returning to state A. Process B performs an input action b followed by an output action $\overline{c}$, returning to state B upon completion.

$$\begin{array}{rcl} A & \stackrel{\text{def}}{=} & a.\overline{b}.A \\ B & \stackrel{\text{def}}{=} & b.\overline{c}.B \end{array}$$

We can think of A of a transducer that transforms the input a to the output b, and similarly for B.

We assume we start with A and B operating concurrently, that is, plugged together, communicating along channel b



In process notation we just separate the processes by a bar "|"

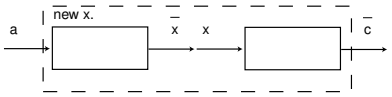
$$A \mid B$$

Now we can have the following sequence of transitions:

$$A \mid B \xrightarrow{a} \overline{b}.A \mid b.\overline{c}.B \longrightarrow A \mid \overline{c}.B \xrightarrow{\overline{c}} A \mid B$$

We have explicitly unfolded B after the first step to make the interaction between $\overline{b}$ and b clear. Note that this synchronization is not an external event, so the transition arrow is unadorned. We call this an *internal action* or *silent action* are write τ.

The second generalization from the sequential processes is to permit name hiding (abstraction). In the example above, we plugged processes A and B together, intuitively connecting the output $\overline{b}$ from A with the input b from B. However, it is still possible to put another process in parallel with A and B that could interact with both of them using b. In order to prohibit such behavior, we can locally bind the name b. We write $\mathsf{new}\, x.P$ for a process with a locally bound name x. Names bound with $\mathsf{new}\, x.P$ are subject to α-conversion (renaming of bound variables) as usual. In a picture:



We might try to express this as

$$\mathsf{new}\, b.A \mid B.$$

However, we have created a new problem: the name b is bound in this expression, but the scope of b does not include the definitions of A and B. In order to avoid this scope violation we parameterize the process definitions by all names that they use, and apply uses of the process identifier with the appropriate local names. We can think of this as a special form of parameter passing or renaming.

$$\begin{aligned} A(x,y) &\stackrel{\text{def}}{=} x.\overline{y}.A\langle x,y\rangle \\ B(y,z) &\stackrel{\text{def}}{=} y.\overline{z}.B\langle y,z\rangle \end{aligned}$$

The process expression can now hygienically refer to locally bound names.

$$\mathsf{new}\, x.A\langle a, x\rangle \mid B\langle x, c\rangle$$

But the definitions of A and B are really the same up to α-conversion, so we can rewrite this also as

$$\mathsf{new}\, x.A\langle a, x\rangle \mid A\langle x, c\rangle$$

This leads to the following language of *concurrent process expressions*.

$$\begin{array}{rrcl} \text{Process Exps} & P & ::= & A\langle a_1, \ldots, a_n\rangle \mid N \mid (P_1 \mid P_2) \mid \mathsf{new}\, x.P \\ \text{Sums} & N & ::= & \alpha.P \mid N_1 + N_2 \mid 0 \\ \text{Action Prefix} & \alpha & ::= & a \mid \overline{a} \mid \tau \end{array}$$

We define the operational semantics of concurrent processes with the set of rules below. In this semantics an action is made explicit in a transition, but matching input/output actions become silent. We use λ to stand for either a or $\overline{a}$ and $\overline{\lambda}$ for $\overline{a}$ or a, respectively.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum}_t \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{React}_t$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{L-Par}_t \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\ \text{R-Par}_t$$

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin \{a, \overline{a}\})}{\mathsf{new}\, a.P \xrightarrow{\alpha} \mathsf{new}\, a.P'}\ \text{Res}_t$$

$$\frac{\{a_1/x_1, \ldots, a_n/x_n\}P_A \xrightarrow{\alpha} P' \quad (A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A)}{A\langle a_1, \ldots, a_n\rangle \xrightarrow{\alpha} P'}\ \text{Ident}_t$$

If we want to examine the interaction of a system with its environment we consider the environment as another *testing process* that is run concurrently with the system whose behavior we wish to examine. In this situation we are mostly interested in silent transitions, since the interactions of a process with its environment are explicit and therefore silent. As example for the above rules, consider the following process expression.

$$P = (\mathsf{new}\, a.((a.Q_1 + b.Q_2) \mid \overline{a}.0)) \mid (\overline{b}.R_1 + \overline{a}.R_2)$$

Note that the output action before R_2 is a different name than a used as the input action to Q_1, the latter being locally quantified. This means there are only two possible τ-transitions.

$$P \longrightarrow (\mathsf{new}\, a.(Q_1 \mid 0)) \mid (\overline{b}.R_1 + \overline{a}.R_2)$$
$$P \longrightarrow (\mathsf{new}\, a.(Q_2 \mid \overline{a}.0)) \mid R_1$$

As another example[1] of this form of concurrent processes, consider two two-way transducers of identical structure.

$$A(a, a', b, b') \stackrel{\text{def}}{=} a.\overline{b}.A\langle a, a', b, b'\rangle + b'.\overline{a'}.A\langle a, a', b, b'\rangle$$

We now compose to instances of this process concurrently, hiding the internal connection between.

$$\mathsf{new}\, b.\mathsf{new}\, b'.(A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle)$$

At first one might suspect this is bisimilar with $A\langle a, a', c, c'\rangle$, which shortcircuits the internal synchronization along b and b'. While we have not formally defined bisimilarity in this new setting, this new composition is in fact buggy: it can deadlock when put in parallel with $\overline{a}.P$, $c.P'$, $\overline{c'}.Q$, $a'.Q'$

$$\begin{aligned}
&\overline{a}.P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle)\\
&\longrightarrow P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{b}.A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle)\\
&\longrightarrow P \mid c.P' \mid Q \mid a'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{b}.A\langle a, a', b, b'\rangle \mid \overline{b'}.A\langle b, b', c, c'\rangle)
\end{aligned}$$

At this point all interactions are blocked and we have a deadlock. This can not happen with the process $A\langle a, a', c, c'\rangle$. It can evolve in different ways but not deadlock in the manner above; here is an example.

$$\begin{aligned}
&\overline{a}.P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid A\langle a, a', c, c'\rangle\\
&\longrightarrow P \mid c.P' \mid \overline{c'}.Q \mid a'.Q' \mid \overline{c}.A\langle a, a', c, c'\rangle\\
&\longrightarrow P \mid P' \mid \overline{c'}.Q \mid a'.Q' \mid A\langle a, a', c, c'\rangle\\
&\longrightarrow P \mid P' \mid Q \mid a'.Q' \mid \overline{a'}.A\langle a, a', c, c'\rangle\\
&\longrightarrow P \mid P' \mid Q \mid Q' \mid A\langle a, a', c, c'\rangle
\end{aligned}$$

The reader should make sure to understand these transition and redesign the composed two-way buffer so that this deadlock situation cannot occur.

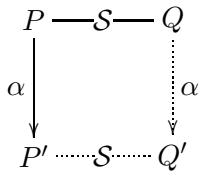
[1] not discussed in lecture

Observational Equivalence for Concurrent Processes .

Next we consider the question of observational equivalence for the calculus of concurrent, communicating processes.

Recall from the last lecture our definition of a *strong simulation* $\mathcal{S}$: If $P \mathcal{S} Q$ and $P \xrightarrow{\alpha} P'$ then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$.

In pictures:



where the solid lines indicate given relationships and the dotted lines indicate the relationships whose existence we have to verify (including the existence of Q'). If such a strong simulation exists, we say that Q strongly simulates P.

Futhermore, we say that two states are *strongly bisimilar* if there is a single relation $\mathcal{S}$ such that both the relation and its converse are strong simulations.

Strong simulation does not distinguish between silent (also called internal or unobservable) transitions τ and observable transitions λ (consisting either of names a or co-names $\overline{a}$). When considering the observable behavior of a process we would like to "ignore" silent transitions to some extent. Of course, this is not entirely possibly, since a silent transition can change from a state with many enabled actions to one with much fewer or different ones. However, we can allow any number of internal actions in order to simulate a transition. We define

$$\begin{array}{lll} P \xrightarrow{\tau^*} P' & \text{iff} & P \xrightarrow{\tau} \cdots \xrightarrow{\tau} P' \\ P \xrightarrow{\tau^* \lambda \tau^*} P' & \text{iff} & P \xrightarrow{\tau^*} P_1 \xrightarrow{\lambda} P_2 \xrightarrow{\tau^*} P' \end{array}$$

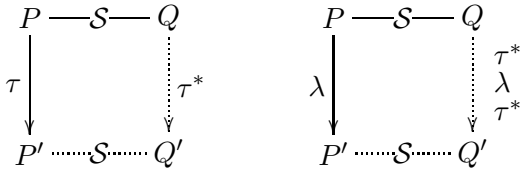
In particular, we always have $P \xrightarrow{\tau^*} P$. Then we say that $\mathcal{S}$ is a *weak simulation* if the following two conditions are satisfied:[2]

(i) If $P \mathcal{S} Q$ and $P \xrightarrow{\tau} P'$
then there exists a Q' such that $Q \xrightarrow{\tau^*} Q'$ and $P' \mathcal{S} Q'$.

(ii) If $P \mathcal{S} Q$ and $P \xrightarrow{\lambda} P'$
then there exists a Q' such that $Q \xrightarrow{\tau^* \lambda \tau^*} Q'$ and $P' \mathcal{S} Q'$.

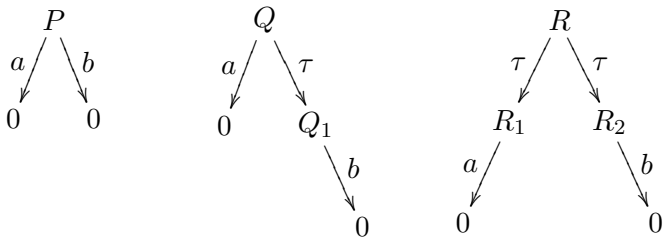
[2]This differs slightly, but I believe insignificantly from Milner's definition.

In pictures:



As before we say that Q *weakly simulates* P if there is a weak simulation $\mathcal{S}$ with $P \; \mathcal{S} \; Q$. We say P and Q are *weakly bisimilar* if there is a relation $\mathcal{S}$ such that both $\mathcal{S}$ and its inverse are weak simulations. We write $P \approx Q$ if P and Q are weakly bisimular.

We can see that the relation of weak bisimulation concentrates on the externally observable behavior. We show some examples that demonstrate processes that are *not* weakly bisimilar.



$$P = a.0 + b.0 \qquad Q = a.0 + \tau.b.0 \qquad R = \tau.a.0 + \tau.b.0$$

Even though P, Q, and R can all weakly simulate each other, no two are weakly bisimilar. As an example, consider P and Q. Then any weak bisimulation must relate P and Q_1, because if $Q \xrightarrow{\tau} Q_1$ then P can match this only by idling (no transition). But $P \xrightarrow{a} 0$ and Q_1 cannot match this step. Therefore P and and Q cannot be weakly bisimilar. Analogous arguments suffice for the other pairs of processes.

As positive examples of weak bisimulation, we have

$$\begin{aligned} a.P &\approx \tau.a.P \\ a.P + \tau.a.P &\approx \tau.a.P \\ a.(b.P + \tau.c.Q) &\approx a.(b.P + \tau.c.Q) + a.c.Q \end{aligned}$$

The reader is encouraged to draw the corresponding transition diagrams. As an example, consider the second equation.

$$Q_1 = a.P + \tau.a.P \quad \text{and} \quad Q_2 = \tau.a.P$$

We relate $Q_1 \; \mathcal{S} \; Q_2$ and $a.P \; \mathcal{S} \; a.P$ and $P \; \mathcal{S} \; P$. In one direction we have

1. $Q_1 \xrightarrow{a} P$ which can be simulated by $Q_2 \xrightarrow{\tau\, a} P$.

2. $Q_1 \xrightarrow{\tau} a.P$ which can be simulated by $Q_2 \xrightarrow{\tau} a.P$.

In the other direction we have

1. $Q_2 \xrightarrow{\tau} a.P$ which can be simulated by $Q_1 \xrightarrow{\tau} a.P$.

Together these cases yield the desired result: $Q_1 \approx Q_2$.

As a final example, we return to the earlier simple transducer.

$$A(x,y) \stackrel{\text{def}}{=} x.\overline{y}.A\langle x,y\rangle$$

Let us compare the two transducers $A\langle a,c\rangle$ and $\mathsf{new}\, x.\, A\langle a,x\rangle \mid A\langle x,c\rangle$. We can easily see that they are not strongly bisimilar, because the latter process can make a silent transition that the first one can not.

But are they weakly bisimilar? It seems both processes can only input a's and produce the same number of c's. Since the intermediate channel x is locally quantified, no other communication with the composed process can take place.

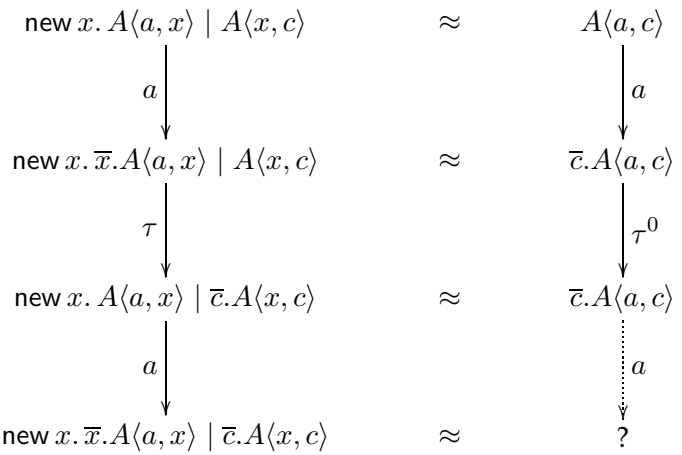
And yet, the two are not weakly bisimilar. To see this, consider the testing process $\overline{a}.\overline{a}.0$. If we put this in parallel with the first process, we obtain

$$\overline{a}.\overline{a}.0 \mid A(a,c) \xrightarrow{\tau} \overline{a}.0 \mid \overline{c}.A\langle a,c\rangle$$

However, if we put it in parallel with the second, we can make further process:

$$\begin{aligned}
&\overline{a}.\overline{a}.0 \mid \mathsf{new}\, x.\, A\langle a,x\rangle \mid A\langle x,c\rangle \\
&\xrightarrow{\tau} \overline{a}.0 \mid \overline{x}.A\langle a,x\rangle \mid A\langle x,c\rangle \\
&\xrightarrow{\tau} \overline{a}.0 \mid A\langle a,x\rangle \mid \overline{c}.A\langle x,c\rangle \\
&\xrightarrow{\tau} 0 \mid \overline{x}.A\langle a,x\rangle \mid \overline{c}.A\langle x,c\rangle \\
&\approx \overline{x}.A\langle a,x\rangle \mid \overline{c}.A\langle x,c\rangle
\end{aligned}$$

More formally, we can prove that they are not bisimilar by assuming that they are and then deriving a contradiction. The contradication can be seen by constructing the following diagram. We construct this by choosing transitions on the left and simulating them on the write. We use τ_0 to stand for idling (= no transition) which is allowed under weak bisimulation.



The last row gives the desired contradiction, since the process $\overline{c}.A\langle a, c\rangle$ cannot exhibit an a transition. Essentially, the transducer on the left has an internal buffer of since one, while the one on the right does not. This shows that in many practical examples, we may need yet other notions of equivalence.

In the next lecture we extend extend the calculus to allow us communication to transmit values, which leads to the π-calculus. Then we will see how a variant of the π-calculus can be embedded in a full-scale language such as Standard ML to offer rich concurrency primitives in addition to functional programming.

Lecture Notes on
The π-Calculus and Concurrent ML

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 26
December 7, 2004

In this lecture we first generalize the calculus of concurrent processes so that values can be transmitted during communication. But our language has no primitive values, so this just reduces to transmitting names along channels that are themselves represented as names. This means that a system of processes can dynamically change its communication structure because connections to processes can be passed as first class values. This is why the resulting language, the π-calculus, is called a calculus of *mobile* and *concurrent* communicating processes. In the second part of the lecture we show how concurrency primitives along the lines of the π-calculus can be embedded in ML, leading to Concurrent ML (CML).

We generalize actions and differentiate them more explicitly into input actions and output actions, since one side of a synchronized communication act has to send and the other to receive a name. We also replace primitive process identifiers and defining equations by process replication $!P$ explained below.

$$
\begin{array}{rrcll}
\text{Action prefixes} & \pi & ::= & a(y) & \text{receive } y \text{ along } a \\
& & \mid & \overline{a}\langle b\rangle & \text{send } b \text{ along } a \\
& & \mid & \tau & \text{unobservable action} \\
\\
\text{Process exps} & P & ::= & N \mid (P_1 \mid P_2) \mid \mathsf{new}\, x.P \mid\, !P & \\
\text{Sums} & N & ::= & 0 \mid N_1 + N_2 \mid \pi.P &
\end{array}
$$

In examples $\pi.0$ is often abbreviated by π. Note that in a summand $a(y).P$, y is a *bound variable* with scope P that stands for the value received along a. On the other hand, $\overline{a}\langle b\rangle.P$ does not bind any variables. Even

though the syntax does not formally distinguish, we use x for binding occurrences of names (subject to renaming), and a and b for non-binding occurrences.

The structural congruence remains the same as before, except that in addition we have $!P \equiv P \mid !P$, that is, a process $!P$ can spawn arbitrarily many copies of itself. For references, we repeat the laws here.

1. Renaming of bound variables (α-conversion)

2. Reordering of terms in a summation

3. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) = (P \mid Q) \mid R$

4. $\mathsf{new}\, x.(P \mid Q) \equiv P \mid \mathsf{new}\, x.Q$ if $x \notin \mathrm{fn}(P)$
   $\mathsf{new}\, x.0 \equiv 0$, $\mathsf{new}\, x.\mathsf{new}\, y.P \equiv \mathsf{new}\, y.\mathsf{new}\, x.P$

5. $!P \equiv P \mid !P$

Before presenting the transition semantics, we consider the following example.

$$P = ((\overline{x}\langle y\rangle.0 + z(w).\overline{w}\langle y\rangle.0) \mid x(u).\overline{u}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0)$$

The middle process can synchronize and communicate with either the first or the last one. Reaction with the first leads to

$$P_1 = (0 \mid \overline{y}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0) \equiv (\overline{y}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0)$$

which cannot transition further. Reaction with the seconds leads to

$$P_1' = ((\overline{x}\langle y\rangle.0 + z(w).\overline{w}\langle y\rangle.0) \mid \overline{z}\langle v\rangle.0 \mid 0)$$

which can step further to

$$P_2' = (\overline{v}\langle y\rangle.0 \mid 0 \mid 0)$$

Next we show the reaction rules in a form which does not make an externally observable action explicit, and exploits structural congruence.

$$\frac{}{\tau.P + N \longrightarrow P}\ \text{Tau}$$

$$\frac{}{(a(x).P + M) \mid (\overline{a}\langle b\rangle.Q + N) \longrightarrow (\{b/x\}P) \mid Q}\ \text{React}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{Par} \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, x.P \longrightarrow \mathsf{new}\, x.P'}\ \text{Res}$$

$$\frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}\ \text{Struct}$$

As a simple example we will model a storage cell that can hold a value and service get and put requests to read and write the cell contents. We first show it using definitions for process identifiers and then rewrite it using process replication.

$$\begin{aligned} C(x, \mathrm{get}, \mathrm{put}) \quad \stackrel{\text{def}}{=} \quad & \overline{get}\langle x\rangle.C\langle x, \mathrm{get}, \mathrm{put}\rangle \\ & + put(y).C\langle y, \mathrm{get}, \mathrm{put}\rangle \end{aligned}$$

We express this in the π-calculus by turning C itself into a name, left-hand side into an input action and occurrences on the right-hand side into an output action.

$$!\, c(x, \mathrm{get}, \mathrm{put}).(\overline{get}\langle x\rangle.\overline{c}\langle x, \mathrm{get}, \mathrm{put}\rangle.0 + put(y).\overline{c}\langle y, \mathrm{get}, \mathrm{put}\rangle.0)$$

We abbreviate this process expression by $!C$. In order to be in the calculus we must be able to receive and send multiple names at once. It is straightforward to add this capability. As an example, consider how to create cell with initial contents 3, write 4 to it, read the cell and then print the contents some output device. Printing a is represented by an output action $\overline{print}\langle a\rangle.0$. We also consider 3 and 4 just as names here.

$$!C \mid \mathsf{new}\, g.\mathsf{new}\, p.\overline{c}\langle 3, g, p\rangle.\overline{p}\langle 4\rangle.g(x).\overline{print}\langle x\rangle.0$$

Note that c and print are the only free names in this expression. Note also that we are creating new names g and p to stand for the channel to get or put a names into the storage cell C. We leave it to the reader as an instructive exercise to simulate the behavior of this expression. It should be clear, however, that we need to use structural equivalence initially to obtain

a copy of C with which we can react after moving the quantifiers of g and p outside.

As a more involved example, consider the following specification of the sieve of Eratosthenes. We start with a stream to produce integers, assuming we have a primitive successor operation on integer names.[1] The idea is to have a channel which sends successive numbers.

$$!count(n, \mathrm{out}).\overline{out}\langle n\rangle.\overline{count}\langle n+1, \mathrm{out}\rangle$$

Second we show a process to filter all multiples of a given prime number from its input stream while producing the output stream. We assume an oracle $(x \bmod p = 0)$ and its negation.

$$\begin{aligned}!filter(p, \mathrm{in}, \mathrm{out}).in(x).((x \bmod p = 0)().\overline{filter}\langle p, \mathrm{in}, \mathrm{out}\rangle.0 \\ + (x \bmod p \neq 0)().\overline{out}(x).\overline{filter}\langle p, \mathrm{in}, \mathrm{out}\rangle.0)\end{aligned}$$

Finally, we come to the process that generates a sequence of prime numbers, starting from the first item of the input channel which should be prime (by invariant).

$$\begin{aligned}&!primes(\mathrm{in}, \mathrm{out}).in(p).\overline{out}\langle p\rangle. \\ &\quad \mathsf{new}\ \mathrm{mid}.(\overline{filter}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \ \mid\ \overline{primes}\langle \mathrm{mid}, \mathrm{out}\rangle.0)\end{aligned}$$

primes establishes a new filtering process for each prime and threads the input stream in into the filter. The first element of the filtered result stream is guaranteed to be prime, so we can invoke the primes process recursively.

At the top level, we start the process with the stream of numbers counting up from 2, the smallest prime. This will generate communication requests $\overline{out}\langle p\rangle$ for each successive prime.

$$\mathsf{new}\ \mathrm{nats}.\overline{count}\langle 2, \mathrm{nats}\rangle \mid \overline{primes}\langle \mathrm{nats}, \mathrm{out}\rangle$$

In this implementation, communication is fully synchronous, that is, both sender and receiver can only move on once the message has been exchanged. Here, this means that the prime numbers are guaranteed to be read in their natural order. If we don't care about the order, we can rewrite the process so that it generates the primes *asynchronously*. For this we use the general transformation of

$$\overline{a}\langle b\rangle.P \quad \Longrightarrow \quad \tau.(\overline{a}\langle b\rangle.0 \mid P)$$

[1] This can also be coded in the π-calculus, but we prefer to avoid this complication here.

which means the computation of P can proceed regardless whether the message b has been received along channel a. In our case, this would be a simple change in the primes generator.

$$!primes(\mathrm{in}, \mathrm{out}).in(p).$$
$$\overline{out}\langle p\rangle.0 \mid \mathsf{new}\,\mathrm{mid}.(\overline{filter}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \mid \overline{primes}\langle \mathrm{mid}, \mathrm{out}\rangle.0)$$

The advantage of an asynchronous calculus is its proximity to a realistic model of computation. On the other hand, synchronous communciation allows for significantly shorter code, because no protocol is needed to make sure messages have been received, and in received in order. Since asynchronous communication is very easily coded here, we stick to Milner's original π-calculus which was synchronous.

In the remainder of this lecture we discuss how Concurrent ML (CML) implements concurrency primitives that heavily borrow from the π-calculus. In CML, channels can carry values (including other channels), communication is synchronous, and execution is concurrent. However, there are also differences. Standard ML is a full-scale programming language, so some idioms that have to be coded painfully in the π-calculus are directly available. Moreover, CML offers another mechanism called *negative acknowledgments*. In this lecture we will not discuss negative acknowledgments and concentrate on the fragment of CML that corresponds most directly to the π-calculus. The examples are drawn from the standard reference:[2]

> John H. Reppy, *Concurrent Programming in ML*, Cambridge University Press, 1999.

We begin with the representation of *names*. In CML they are represented by the type τ `chan` that carries values of type τ. We show the relevant portion of the signature for the structure `CML`.

```
type 'a chan
val channel : unit -> 'a chan
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

The `send` and `recv` operations are *synchronous* which means that a call `send (a, v)` will block until there is a matching `recv (a)` in another thread of computation and the two rendezvous. We will see later that `send` and `recv` are actually definable in terms of some lower-level constructs.

[2] See also `http://people.cs.uchicago.edu/~jhr/cml/`.

What we called a process in the π-calculus is represented as a *thread* of computation in CML. They are called threads to emphasize their relatively lightweight nature. Also, they are executing with shared memory (the Standard ML heap), even though the model of communication is *message passing*. This imposes a discipline upon the programmer not to resort to possibly dangerous and inefficient use of mutable references in shared memory and use message passing instead.

The relevant part of the CML signature is reproduced below. In this lecture we will not use `thread_id` which is only necessary for other styles of concurrent programming.

```
type thread_id
val spawn : (unit -> unit) -> thread_id
val exit : unit -> 'a
```

Even without non-deterministic choice, that is, the sums from the π-calculus, we can now write some interesting concurrent programs. The example we use here is the sieve of Eratosthenes presented in the π-calculus in the last lecture. The pattern of programming this examples and other related programs in CML is the following: a function will accept a parameter, spawn a process, and return one or more channels for communication with the process it spawned.

The first example is a counter process that produces a sequence of integers counting upwards from some number n. The implementation takes n as an argument, creates an output channel, defines a function which will be the looping thread, and then spawns the thread before returning the channel.

```
(* val counter : int -> int CML.chan *)
fun counter (n) =
    let
      val outCh = CML.channel ()
      fun loop (n) = (CML.send (outCh, n); loop (n+1))
    in
      CML.spawn (fn () => loop n);
      outCh
    end
```

The internal state of the process is not stored in a reference, but as the argument of the `loop` function which runs in the counter thread.

Next we define a function `filter` which takes a prime number `p` as an argument, together with an input channel `inCh`, spawns a new filtering process and returns an output channel which returns the result of removing all multiples of `p` from the input channel.

```
(* val filter : int * int CML.chan -> int CML.chan *)
fun filter (p, inCh) =
    let
      val outCh = CML.channel ()
      fun loop () =
          let val i = CML.recv inCh
          in
            if i mod p <> 0
              then CML.send (outCh, i)
            else ();
            loop ()
          end
    in
      CML.spawn (fn () => loop ());
      outCh
    end
```

Finally, the `sieve` function which returns a channel along which an external thread can receive successive prime numbers. It follows the same structure as the functions above.

```
(* val sieve : unit -> int CML.chan *)
fun sieve () =
    let
      val primes = CML.channel ()
      fun loop ch =
          let
            val p = CML.recv ch
            val _ = CML.send (primes, p)
            val mid = filter (p, ch)
          in
            loop (mid)
          end
    in
      CML.spawn (fn () => loop (counter 2));
      primes
    end
```

When `sieve` is creates a new channel and then spawns a process that will produces prime numbers along this channel. It also spawns a process to enumerate positive integers, starting with 2 and counting upwards. At this point it blocks, however, until someone tries to read the first prime number from its output channel. Once that rendezvous has taken place, it spawns a new thread to filter multiples of the last prime produced with `filter (p, ch)` and uses that as its input thread.

To produce a list of the first n prime numbers, we successively communicate with the main thread spawned by the call to `sieve`.

```
(* val primes : int -> int list *)
fun primes (n) =
    let
      val ch = sieve ()
      fun loop (0, l) = List.rev l
        | loop (n, l) = loop (n-1, CML.recv(ch)::l)
    in
      loop (n, nil)
    end
```

For non-deterministic choice during synchronization, we need a new notion in CML which is called an *event*. Events are values that we can synchronize on, which will block the current thread. Event combinators will allow us to represent non-deterministic choice. The simplest forms of events are *receive* and *send* events. When synchronized, they will block until the rendezvous along a channel has happened.

```
type 'a event
val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
val never : 'a event
val alwaysEvt : 'a -> 'a event
val wrap : 'a event * ('a -> 'b) -> 'b event
val choose : 'a event list -> 'a event
val sync : 'a event -> 'a
```

Synchronization is achieved with the function `sync`. For example, the earlier `send` function can be defined as

```
val send = fn (a,x) => sync (sendEvt (a,x))
```

that is, `val send = sync o sendEvt`.

We do not use `alwaysEvt` here, but its meaning should be clear: it corresponds to a τ action returning a value without any communication.

`choose [`$v_1, \ldots, v_n$`]` for event values $v_1, \ldots, v_n$ corresponds to a sum $N_1 + \cdots + N_n$. In particular, `choose []` will block and can never proceed, while `choose [`v`]` should be equivalent to v.

`wrap (`v`,` f`)` provides a function v to be called on the result of synchronizing v. This is needed because different actions may be taken in the different branches of a `choose`. It is typical that each primitive receive or send event in a non-deterministic choice is wrapped with a function that indicates the action to be taken upon the synchronization with the event.

As an example we use the implementation of a storage cell via a concurrent process. This is an implementation of the following signature.

```
signature CELL =
sig
  type 'a cell
  val cell : 'a -> 'a cell
  val get : 'a cell -> 'a
  val put : 'a cell * 'a -> unit
end;
```

In this example, creating a channel returns two channels for communication with the spawned thread: one to read the contents of the cell, and one to write the contents of the cell. It is up to the client program to make sure the calls to `get` and `put` are organized in a way that does not create incorrect interference in case different threads want to use the cell.

```
structure Cell' :> CELL =
struct
datatype 'a cell =
  CELL of 'a CML.chan * 'a CML.chan
fun cell x =
    let
      val getCh = CML.channel ()
      val putCh = CML.channel ()
      fun loop x = CML.synch (
        CML.choose [CML.wrap (CML.sendEvt (getCh, x),
                              fn () => loop x),
                    CML.wrap (CML.recvEvt putCh,
                              fn x' => loop x')])
    in
      CML.spawn (fn () => loop x);
      CELL (getCh, putCh)
    end
fun get (CELL(getCh, _)) = CML.recv getCh
fun put (CELL(_, putCh), x) = CML.send (putCh, x)
end;
```

This concludes our treatment of the high-level features of CML. Next we will sketch a formal semantics that accounts for concurrency and synchronization. The most useful basis is the C-machine, which makes a continuation stack explicit. This allows us to easily talk about blocked processes or synchronization. The semantics is a simplified version of the one presented in Reppy's book, because we do not have to handle negative acknowledgments. Also, the notation is more consistent with our earlier development.

First, we need to introduce channels. We denote them by a, following the π-calculus. Channels are typed $a : \tau\,\mathsf{chan}$ for types τ. During the evaluation, new channels will be created and have to be carried along as a *channel environment*. This reminiscent of thunks, or memory in other evaluation models we have discussed. These channels are global, that is, shared across the whole process state. Finally we have the state s of individual thread, which are as in the C-machine.

$$\begin{array}{rrcl} \text{Channel env} & \mathcal{N} & ::= & \cdot \mid \mathcal{N}, a\,\mathsf{chan} \\ \text{Machine state} & P & ::= & \cdot \mid P, s \\ \text{Thread state} & s & ::= & K > e \mid K < v \end{array}$$

In order to write rules more compactly, we allow the silent re-ordering of threads in a machine state. This does imply any scheduling strategy.

We have two judgments for the operational semantics

$s \mapsto s'$	Thread steps from s to s'
$(\mathcal{N} \vdash P) \mapsto (\mathcal{N}' \vdash P')$	Machine steps from P to P'

In the latter case we know that $\mathcal{N}'$ is either $\mathcal{N}$ or contains one additional channel that may have been created. The first judgment, $s \mapsto s'$ is exactly as it was before in the C-machine. We have one general rule

$$\frac{s \mapsto s'}{(\mathcal{N} \vdash P, s) \mapsto (\mathcal{N} \vdash P, s')}$$

We now define the new constructs, one by one.

Channels. Channels are created with the `channel` function. They are value.

$$\frac{}{a \ \mathsf{value}} \qquad \frac{(a \ \mathsf{chan} \notin \mathcal{N})}{(\mathcal{N} \vdash P, K > \texttt{channel}\,()) \mapsto (\mathcal{N}, a \ \mathsf{chan} \vdash P, K < a)}$$

We do not define the semantics of the `send` and `recv` functions because they are definable.

Threads. New threads are created with the `spawn` function. We ignore here the `thread_id` type and return a unit element instead.

$$\frac{}{(\mathcal{N} \vdash P, K > \texttt{spawn}\, v) \mapsto (\mathcal{N} \vdash P, \bullet > v\,(), K < ())}$$

$$\frac{}{(\mathcal{N} \vdash P, K > \texttt{exit}\,()) \mapsto (\mathcal{N} \vdash P)}$$

Recall that even though we write the relevant thread among P last, it could in fact occur anywhere by our convention that the order of the threads is irrelevant.

Finally, we come to events. We make one minor change to make them syntactically easier to handle. Instead of `choose` to take an arbitrary list of events, we have two constructs:

```
val choose : 'a event * 'a event -> 'a event
val never : 'a event
```

Events must be values in this implementation, because they must become arguments to the synchronization function `sync`.

$$\frac{v \ \text{value}}{\texttt{sendEvt}(a, v) \ \text{value}} \qquad \frac{}{\texttt{recvEvt}(a) \ \text{value}} \qquad \frac{v \ \text{value}}{\texttt{always}(v) \ \text{value}}$$

$$\frac{v_1 \ \text{value} \quad v_2 \ \text{value}}{\texttt{choose}(v_1, v_2) \ \text{value}} \qquad \frac{}{\texttt{never} \ \text{value}}$$

$$\frac{v_1 \ \text{value} \quad v_2 \ \text{value}}{\texttt{wrap}(v_1, v_2) \ \text{value}}$$

From these value definitions one can straightforwardly derive the rules that evaluate subexpressions. Interestingly, there only two new rules for the operational semantics: for two-way synchronization (corresponding to a value being sent) and one-way synchronization (corresponding to a τ-action with a value). This requires two new judgments, $(v, v') \rightsquigarrow (e, e')$ and $v \rightsquigarrow e$. We leave the one-way synchronization as an exercise and show the details of two-way synchronization.

$$\frac{(v, v') \rightsquigarrow (e, e')}{(\mathcal{N} \vdash P, K > \texttt{sync}(v), K > \texttt{sync}(v')) \mapsto (\mathcal{N} \vdash P, K > e, K > e')} \ \text{R}_2$$

$$\frac{v \rightsquigarrow e}{(\mathcal{N} \vdash P, K > \texttt{sync}(v)) \mapsto (\mathcal{N} \vdash P, K > e)} \ \text{R}_1$$

The judgment $(v, v') \rightsquigarrow (e, e')$ means that v and v' can rendezvous, returning expression e to the first thread and e' to the second thread. We show the rules for it in turn, considering each event combinator. We presuppose that subexpressions marked v are indeed values, without checking this explicitly with the v value judgment.

Send and receive events. This is the base case. The sending thread continues with the unit element, while the receiving thread continues with the value carried along the channel a.

$$\frac{}{(\texttt{sendEvt}(a, v), \texttt{recvEvt}(a)) \rightsquigarrow ((), v)} \ sr$$

$$\frac{}{(\texttt{recvEvt}(a), \texttt{sendEvt}(a, v)) \rightsquigarrow (v, ())} \ rs$$

Choice events. There are no rules to synchronize on `never` events, and there are four rules for the binary `choose` event.

$$\frac{(v_1, v') \rightsquigarrow (e, e')}{(\texttt{choose}(v_1, v_2), v') \rightsquigarrow (e, e')}\ c_1^l \qquad \frac{(v_2, v') \rightsquigarrow (e, e')}{(\texttt{choose}(v_1, v_2), v') \rightsquigarrow (e, e')}\ c_2^l$$

$$\frac{(v, v_1') \rightsquigarrow (e, e')}{(v, \texttt{choose}(v_1', v_2')) \rightsquigarrow (e, e')}\ c_1^r \qquad \frac{(v, v_2') \rightsquigarrow (e, e')}{(v, \texttt{choose}(v_1', v_2')) \rightsquigarrow (e, e')}\ c_2^r$$

Wrap events. Finally we have `wrap` events that construct bigger expressions, to be evaluated if synchronization selects the corresponding event. This is way synchronization returns an expression, to be evaluated further, rather than a value.

$$\frac{(v_1, v') \rightsquigarrow (e_1, e')}{(\texttt{wrap}(v_1, v_2), v') \rightsquigarrow (v_2\, e_1, e')}\ w^l$$

$$\frac{(v, v_1') \rightsquigarrow (e, e_1')}{(v, \texttt{wrap}(v_1', v_2')) \rightsquigarrow (e, v_2'\, e_1')}\ w^r$$

With the typing rules derived from the CML signature and the operational semantics, it is straightforward to prove a type preservation result. The only complication is presented by names, since they are created dynamically. But we have already seen the solution to a very similar problem when dealing with mutable references (since locations l are also created dynamically), so no new concepts are required.

Progress is more difficult. The straightforward statement of the progress theorem would be false, since the type system does not track whether processes can in fact deadlock. Also, we would have to re-think what nontermination means, because some processes might run forever, while others terminate, while yet others block. We will not explore this further, but it would clearly be worthwhile to verify that any thread can either progress, exit, return a final value, or block on an event. This means that there are no "unexpected" violations of progress. Along similar lines, it would be very interesting to consider type systems in which concurrency and communication is tracked to the extent that a potential deadlock would be a type error! This is currently an active area of research.