Supplementary Notes on Inductive Definitions

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 2
August 29, 2002

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. 4.1]. 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

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

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

Rewritten as inference rules we have:

$$\frac{}{\varepsilon\ M} \tag{1}$$

$$\frac{s_1\ M \qquad s_2\ M}{s_1\ s_2\ M} \tag{2}$$

$$\frac{s\ M}{(\,s\,)\ M} \tag{3}$$

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 (1), (2), and (3).*

Based on this interpretation we can prove properties of strings in the syntactic category M by rule induction. 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 1) Then $s = \varepsilon$.

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

(Rule 2) 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 $s = s_1\ s_2$

(Rule 3) Then $s = (\,s'\,)$.

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

■

The grammar we gave, unfortunately, is ambiguous [Ch. 4.2]. 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. 5.1] 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. In order to illustrate the concept of simultaneous inductive definition, we use two non-terminals L and N, where the category L corresponds to M, while N is an auxiliary non-terminal.

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

One can think of L as a list of parenthesized expressions, while N is a single, non-empty parenthesized expression. This is readily translated into an inductive definition via inference rules.

$$\frac{}{\varepsilon\ L} \tag{4}$$

$$\frac{s_1\ N \qquad s_2\ L}{s_1\ s_2\ L} \tag{5}$$

$$\frac{s\ L}{(\,s\,)\ N} \tag{6}$$

Note that the definitions of $s\ L$ and $s\ N$ depend on each other. This is an example of a *simultaneous inductive definition*.

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

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" statement 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.

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 4) Then $s_1 = \varepsilon$.

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

(Rule 5) Then $s_1 = s_{11}\ s_{12}$.

$s_{11}\ N$	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 (5)

■

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 1) Then $s = \varepsilon$.

$s\ L$	By rule (4) since $s = \varepsilon$

(Rule 2) 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)

(Rule 3) Then $s = (\ s'\ )$.

$s'\ M$	Subderivation
$s'\ L$	By i.h.
$(\ s'\ )\ N$	By rule (6)
$\varepsilon\ L$	By rule (4)
$(\ s'\ )\ L$	By rule (5) and $(\ s'\ )\ \varepsilon = (\ s'\ )$

■

The right-to-left direction presents a slightly different problem, namely that the statement "*If* $s\ L$ *then* $s\ M$" does not speak about $s\ N$, even though L and N depend on each other. In such a situation we typically have to generalize the induction hypothesis to also assert an appropriate property of the auxiliary judgments ($s\ N$, in this case). This is the first alternative proof below. The second alternative proof uses a proof principle called inversion, closely related to induction. We present both proofs to illustrate both techniques.

Lemma 4 (First Alternative, Using Generalization)

1. *If* $s\ L$ *then* $s\ M$.

2. *If* $s\ N$ *then* $s\ M$.

Proof: By simultaneous induction on the given derivations. There are two cases to consider for part 1 and one case for part 2.

(Rule 4) Then $s = \varepsilon$.

$s\ M$	By rule (1) since $s = \varepsilon$

(Rule 5) Then $s = s_1\, s_2$.

$s_1\ N$	Subderivation
$s_2\ L$	Subderivation
$s_1\ M$	By i.h.(2)
$s_2\ M$	By i.h.(1)
$s_1\, s_2\ M$	By rule (2)

(Rule 6) Then $s = (\, s'\,)$.

$s'\ L$	Subderivation
$s'\ M$	By i.h.(1)
$(\, s'\,)\ M$	By rule (3)

■

For this particular lemma, we could have avoided the generalization and instead proven (1) directly by using a new form of argument called *inversion*. Since it is an important principle, we will also show this alternative proof.

Lemma 4 (Second Alternative, Using Inversion)
If $s\ L$ then $s\ M$

Proof: By induction on the given derivation. Note the there are only two cases to consider here instead of three, because there are only two rules whose conclusion has the form $s\ L$.

(Rule 4) Then $s = \varepsilon$.

$s\ M$	By rule (1) since $s = \varepsilon$

(Rule 5) Then $s = s_1\, s_2$.

$s_1\ N$	Subderivation
$s_1 = (\, s_1'\,)$ and $s_1'\ L$ for some s_1'	By inversion
$s_1'\ M$	By i.h.
$(\, s_1'\,)\ M$	By rule (3)
$s_2\ L$	Subderivation
$s_2\ M$	By i.h.
$(\, s_1'\,)\ s_2\ M$	By rule (2)
$s\ M$	Since $s = s_1\, s_2 = (\, s_1'\,)\, s_2$

In this last case, the first line reminds us that we have a subderivation of $s_1\ N$. By examining all inference rules we can see that there is exactly one rule that has a conclusion of this form, namely rule (6). Therefore $s_1\ N$ *must* have been inferred with that rule, and s_1 must be equal to $(\ s_1'\ )$ for some s_1' such that $s_1'\ L$. Moreover, the derivation of $s_1'\ L$ is a subderivation of the one we started with and we can therefore apply the induction hypothesis to it. The rest of the proof is routine. ■

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 2), 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 such as the first alternative of Lemma 4 in which the nature of the mutually recursive judgments suggested a generalization.

Finding a counterexample greatly varies in difficulty. Mostly, in this course, counterexample 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. In no other case can use 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.

Supplementary Notes on Abstract Syntax

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 3
September 3, 2002

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. 1.2.1].

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 and 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. 4.1] 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.

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

$$\frac{s\ \mathsf{D}}{s\ \mathsf{N}} \qquad \frac{s_1\ \mathsf{N} \qquad s_2\ \mathsf{D}}{s_1\,s_2\ \mathsf{N}}$$

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

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

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

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 1.1.1], we take them here as primitive mathematical objects.

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

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.

$$\frac{k \text{ nat}}{\texttt{num}(k) \text{ expr}}$$

$$\frac{t_1 \text{ expr} \qquad t_2 \text{ expr}}{\texttt{plus}(t_1, t_2) \text{ expr}}$$

$$\frac{t_1 \text{ expr} \qquad t_2 \text{ expr}}{\texttt{times}(t_1, t_2) \text{ expr}}$$

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.

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

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

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

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

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

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 $s \ \mathsf{E} \longleftrightarrow t \text{ expr}$ *then* $s \ \mathsf{E}$ *and* $t \text{ expr}$.

(ii) If $s \ \mathsf{E}$ *then there exists a* t *such that* $s \ \mathsf{E} \longleftrightarrow t \text{ expr}$.

Proof: For each part, by rule induction on the given derivation. 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. 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 3 (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 4 (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,

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

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

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

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

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

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

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

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

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

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

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

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

and allow for expressions

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

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 again 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.

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

As a concrete example, consider the string

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

which, in abstract syntax, would be represented as

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

Supplementary Notes on Static and Dynamic Semantics

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 4
September 6, 2002

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 serve 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

$$x.t \equiv y.\{y/x\}t$$

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:

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

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]

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

[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

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

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

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

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

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.

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

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 $u \equiv u'$, $t \equiv t'$, and $\{u/x\}t$ and $\{u'/x\}t'$ are both defined, then $\{u/x\}t \equiv \{u'/x\}t'$.

(ii) Given u, x, and t, then there always exists a $t' \equiv t$ 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, $\mathrm{FV}(e) = \{\}$. In order to specify this inductively, we 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:

> *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.[2] We write '$\cdot$' for an empty collection of assumptions, and we abbreviate $\cdot, x{:}\mathtt{nat}$ by $x{:}\mathtt{nat}$.

[2] 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.

The judgment is defined by the following rules.

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

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

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

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

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

This generalized statement ca 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 $\mathtt{num}(k)$. 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 although they do not specify how to compute the value precisely.

The first way[3] 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

[3] suggested by a student in class

the value of a variable is uniquely determined.

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

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

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

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.

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

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

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

In the next lecture we discuss how these two alternatives for the operational semantics are related, and how they are related to the typing judgment.

Supplementary Notes on A Functional Language

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 5
September 10, 2002

In this lecture we first show the equivalence of the two styles of operational semantics: a *substitution semantics* and an *environment semantics*. We then proceed to extend our expression language to include booleans and functions.

We first recall the environment semantics, presented here as a particular form of evaluation semantics [Ch. 7.2]. The basic judgment is

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

Recall that this is a hypothetical judgment with assumptions $x_i \Downarrow v_i$. 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. Here we assume some primitive operators $\mathtt{o}$ (such as `plus` and `times`) and their mathematical counterparts f_o. For simplicity, we just write binary operators here.

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

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

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

The alternative semantics uses substitution instead of environments. For this judgment we evaluate only closed terms, so no hypothetical judg-

ment is needed.

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

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

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

We show each direction of the translation between the two systems separately. In the first direction we assume $\cdot \vdash e \Downarrow v$ and we want to show $e \Downarrow v$. A direct proof by induction is suspect, because the environment will in general not be empty in the derivation of $\cdot \vdash e \Downarrow v$. In particular, the second premise of $e.let$ adds a new assumption, which prevents us from using the induction hypothesis.

In order to generalize the induction hypothesis, we need to figure out what corresponds to $\eta \vdash e \Downarrow v$ in the substitution semantics. From the definition of the semantics we can see that an environment is a "postponed" substitution: rather than carrying out the substitution for each variable as we encounter it, we look up the variable at the end when we see it. Formalizing this intuition is the key to the proof. We define the translation from an environment to a simultaneous substitution [Ch. 5.3]

$$(x_1 \Downarrow v_1, \ldots, x_n \Downarrow v_n)^* = (v_1/x_1, \ldots, v_n/x_n)$$

Then we generalize to account for environments.

Lemma 1
If $\eta \vdash e \Downarrow v$ *then* $\{\eta^*\}e \Downarrow v$.

Proof: By rule induction on the given derivation. Recall that values v always have the form $\mathsf{num}(k)$ for some k, so $v \Downarrow v$ for any value v by rule $s.num$.

Case: (Rule $e.var$**)** Then $e = x$.

$x \Downarrow v \in \eta$	Condition of $e.var$
$v/x \in \eta^*$	By definition of η^*
$\{\eta^*\}x = v$	By definition of substitution
$v \Downarrow v$	By definition of v and rule $s.num$

Case: (Rule $e.num$**)** Then $e = \mathsf{num}(k) = v$.

$\mathsf{num}(k) \Downarrow \mathsf{num}(k)$	By rule $s.num$

Case: (Rule $e.o$**)** Then $e = \mathsf{o}(e_1, e_2)$.

$\eta \vdash e_1 \Downarrow \mathsf{num}(k_1)$	Subderivation
$\eta \vdash e_2 \Downarrow \mathsf{num}(k_2)$	Subderivation
$f_o(k_1, k_2) = k$	Given condition
$\{\eta^*\}e_1 \Downarrow \mathsf{num}(k_1)$	By i.h.
$\{\eta^*\}e_2 \Downarrow \mathsf{num}(k_2)$	By i.h.
$\mathsf{o}(\{\eta^*\}e_1, \{\eta^*\}e_2) \Downarrow \mathsf{num}(k)$	By rule $s.o$
$\{\eta^*\}\mathsf{o}(e_1, e_2) \Downarrow \mathsf{num}(k)$	By definition of substitution

Case: (Rule $e.let$**)** Then $e = \mathtt{let}(e_1, x.e_2)$ and $v = v_2$.

$\eta \vdash e_1 \Downarrow v_1$	Subderivation
$\eta, x{\Downarrow}v_1 \vdash e_2 \Downarrow v_2$	Subderivation
$\{\eta^*\}e_1 \Downarrow v_1$	By i.h.
$(\eta, x{\Downarrow}v_1)^* = (\eta^*, v_1/x)$	By definition of $(\,)^*$
$\{\eta^*, v_1/x\}e_2 \Downarrow v_2$	By i.h.
$\{v_1/x\}(\{\eta^*\}e_2) \Downarrow v_2$	By properties of simultaneous substitution
$\mathtt{let}(\{\eta^*\}e_1, x.\{\eta^*\}e_2)$	By rule $s.let$
$\{\eta^*\}\mathtt{let}(e_1, x.e_2)$	By definition of substitution

■

In the last case we need two properties that connects simultaneous substitution and the "single" substitution $\{v_1/s\}$. They are (a) that the order of the definition of variables in a simultaneous substitution does not matter, and (b) that

$$\{v_1/x_1\}(\{v_2/x_2, \ldots, v_n/x_n\}e) = \{v_1/x_1, v_2/x_2, \ldots, v_n/x_n\}e.$$

These properties hold under the assumption that all the x_i are distinct and that all $v_1, v_2, \ldots, v_n$ are closed, which is known in our case.

In lecture we proceeded slightly differently. Although the essential idea we were converging on was the same, we were getting to a lemma which asserted that $\eta \vdash e \Downarrow v$ then $\cdot \vdash \{\eta^*\}e \Downarrow v$ with a derivation of equal length. The above proof is somewhat more economical.

The other direction is quite a bit tricker to generalize correctly.

Lemma 2
If $e \Downarrow v$ *and* $e = \{\eta^*\}e'$ *then* $\eta \vdash e' \Downarrow v$.

Proof: The proof is by rule induction on the derivation of $e \Downarrow v$

Case: (Rule $s.num$**)** Then we have to consider two subcases, depending on whether $e' = x$ for some variable x, or $e' = \mathsf{num}(k)$ for some k.

Subcase: (Rule $s.num$ **and** $e' = x$**)** Then $x \Downarrow v \in \eta$ in order for $e = \{\eta^*\}x \Downarrow v$ and hence $\eta \vdash x \Downarrow v$ by rule $e.var$.

Subcase: (Rule $s.num$ **and** $e' = \mathsf{num}(k)$**)** In that case $v = \mathsf{num}(k)$, so we can use rule $e.num$.

Case: (Rule $s.o$**)** Then $e = \mathsf{o}(e_1, e_2) = \{\eta^*\}e'$.

$e' = \mathsf{o}(e'_1, e'_2)$ with	
$e_1 = \{\eta^*\}e'_1$ and $e_2 = \{\eta^*\}e'_2$	By definition of substitution
$e_1 \Downarrow \mathsf{num}(k_1)$	Subderivation
$e_2 \Downarrow \mathsf{num}(k_2)$	Subderivation
$f_o(k_1, k_2) = k$	Given condition
$\eta \vdash e'_1 \Downarrow \mathsf{num}(k_1)$	By i.h.
$\eta \vdash e'_2 \Downarrow \mathsf{num}(k_2)$	By i.h.
$\eta \vdash \mathsf{o}(e'_1, e'_2) \Downarrow k$	By rule $e.o$

Case: (Rule $s.let$**)** Then $e = \mathtt{let}(e_1, x.e_2) = \{\eta^*\}e'$ and $v = v_2$.

$e' = \mathtt{let}(e'_1, x.e'_2)$ with	
$e_1 = \{\eta^*\}e'_1$ and $e_2 = \{\eta^*\}e'_2$ and	
x not defined in η	By definition of substitution
$e_1 \Downarrow v_1$	Subderivation
$\eta \vdash e'_1 \Downarrow v_1$	By i.h.
$\{v_1/x\}e_2 \Downarrow v_2$	Subderivation
$\{v_1/x\}e_2 = \{v_1/x\}(\{\eta^*\}e'_2) = \{\eta^*, v_1/x\}e'_2$	Property of substitution
$\{(\eta, x \Downarrow v_1)^*\}e'_2 \Downarrow v_2$	By definition of $(\,)^*$
$\eta, x \Downarrow v_1 \vdash e'_2 \Downarrow v_2$	By i.h.
$\eta \vdash \mathtt{let}(e'_1, x.e'_2) \Downarrow v_2$	By rule $e.let$

■

Now we can prove our main theorem.

Theorem 3 (Equivalence of Environment and Substitution Semantics)
(i) If $\cdot \vdash e \Downarrow v$ *then* $e \Downarrow v$

(ii) If $e \Downarrow v$ *then* $\cdot \vdash e \Downarrow v$.

Proof: Part *(i)* follows immediately from the first lemma with $\eta = \cdot$, the empty environment.

Part *(ii)* follows from the second lemma by using the empty environment for η and e for e', which is correct since $e = \{\cdot\}e$. ■

We now proceed with the introduction of MinML. The treatment here is somewhat cursory; see [Ch. 8] for additional material. Roughly speaking, MinML arises from the arithmetic expression language by adding booleans and recursive functions. These recursive 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.

Types	τ	$::=$	$\texttt{int} \mid \texttt{bool} \mid \texttt{arrow}(\tau_1, \tau_2)$
Integers	n	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
Primops	o	$::=$	$\texttt{plus} \mid \texttt{minus} \mid \texttt{times} \mid \texttt{negate}$
		$\mid$	$\texttt{equals} \mid \texttt{lessthan}$
Expressions	e	$::=$	$\texttt{num}(n) \mid o(e_1, \ldots, e_n)$
		$\mid$	$\texttt{true} \mid \texttt{false} \mid \texttt{if}(e, e_1, e_2)$
		$\mid$	$\texttt{let}(e_1, x.e_2)$
		$\mid$	$\texttt{fun}(\tau_1, \tau_2, f.x.e) \mid \texttt{apply}(e_1, e_2)$
		$\mid$	x

Note that, unlike ML, the `fun`-expression binds both f (the function) and x (the argument). It does not define f in the rest of the program, only in the function body e in order to allow a recursive call. For example, the concrete syntax function

```
fun p(x:int):int is if x = 0
                    then 1
                    else 2 * p(x-1) fi end
```

is represented by

$$\begin{array}{l}\texttt{fun}(\texttt{int},\texttt{int},p.x.\texttt{if}(\texttt{equals}(x,\texttt{num}(0))\\ \qquad\qquad\qquad\qquad\texttt{num}(1),\\ \qquad\qquad\qquad\qquad\texttt{times}(\texttt{num}(2),\texttt{apply}(p,\texttt{minus}(x,\texttt{num}(1)).))))\end{array}$$

This is a naive implementation of $p(x) = 2^x$ for $x \geq 0$. If $x < 0$, it will simply not terminate.

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

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

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

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

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

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

$$\frac{\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e) : \texttt{arrow}(\tau_1, \tau_2)}\ \textit{FunTyp}$$

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

We specify the operational semantics as a *structured operational semantics* also called a *small-step semantics*. The reason for this style of specification is that the evaluation semantics (also called big-step semantics) we used so far makes it difficult to talk about non-termination and the individual steps during evaluation, because it is slightly too abstract.

So we define two basic judgments

(i) $e \mapsto e'$ which expresses that e steps to e', and

(ii) e value which expresses that e is a value (written v)

The idea is that, given a closed, well-typed expression e_1, computation proceeds step-by-step until it reaches a value:

$$e_1 \mapsto e_2 \mapsto \cdots \mapsto v$$

where v value. We will eventually prove the following three important properties, which guide us in the design of the rules

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

Note that for all three properties we are only interested in closed, well-typed expressions.

When presenting the operational semantics, we proceed type by type.

Integers This is straightforward. First, integers themselves are values.

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

Second, we evaluate the arguments to a primitive operation from left to right, and apply the operation once all arguments have been evaluated.

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

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

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

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

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

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 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 (recursive) function by itself is a value.

$$\frac{}{\texttt{fun}(\tau_1, \tau_2, f.x.e) \text{ value}}$$

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.

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

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\texttt{apply}(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e}$$

Supplementary Notes on Type Safety

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 6
September 12, 2002

In this lecture we discuss and prove several language properties of MinML that connect the type system to the operational semantics. In particular, we will show

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 languages 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 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 un-

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

fortunate for a language which could quite easily be deterministic, and is intended to be used deterministically.

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

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 value 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.

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

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e_1)) \quad v_2 \text{ value}}{\texttt{apply}(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e_1}$$

$\cdot \vdash \texttt{apply}(v_1, v_2) : \tau$	Assumption
$\cdot \vdash v_1 : \texttt{arrow}(\tau', \tau)$ and $\cdot \vdash v_2 : \tau'$ for some τ'	By inversion
$\cdot \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e_1) : \texttt{arrow}(\tau', \tau)$	By definition of v_1
f:$\texttt{arrow}(\tau', \tau), x$:$\tau' \vdash e_1 : \tau$ and $\tau_1 = \tau'$ and $\tau_2 = \tau$	By inversion
f:$\texttt{arrow}(\tau', \tau) \vdash \{v_2/x\}e_1 : \tau$	By value substitution property
$\cdot \vdash \{v_1/f\}\{v_2/x\}e_1 : \tau$	By value 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 value substitution property. The latter states that substuting a (closed) value of type τ for a variable of type τ in an expression of type τ' preserves the type of that expression as τ'.

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,

$$\texttt{apply}(\texttt{fun}(\texttt{int}, \texttt{int}, f.x.\texttt{apply}(f, x)), \texttt{num}(0))$$

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 3 (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* $v = \texttt{fun}(\tau_1, \tau_2, f.x.e)$ *for some* $f.x.e$.

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: $v = \texttt{fun}(\tau_1, \tau_2, f.x.e)$**.** 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. 9.2]. Now we can prove the progress theorem.

Theorem 4 (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

$$\frac{f{:}\texttt{arrow}(\tau_1, \tau_2), x{:}\tau_1 \vdash e : \tau_2}{\cdot \vdash \texttt{fun}(\tau_1, \tau_2, f.x.e) : \texttt{arrow}(\tau_1, \tau_2)}\ \textit{FunTyp}$$

Then $\texttt{fun}(\tau_1, \tau_2, f.x.e)$ value.

Case

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

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 value	By i.h.
$e_1 \mapsto e_1'$	First subcase
$\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)$	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
$\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1, e_2')$	By rule (since e_1 value)
e_2 value	Second subsubcase
$e_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e_1')$	By value inversion
$\texttt{apply}(e_1, e_2) \mapsto \{e_1/f\}\{e_2/x\}e_1'$	By rule (since e_2 value)

■

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.

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

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

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\texttt{apply}(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e}\ cbv.r$$

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.

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

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e))}{\texttt{apply}(v_1, e_2) \mapsto \{v_1/f\}\{e_2/x\}e}\ cbn.r$$

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.

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

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

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2\ \textsf{value}}{\texttt{apply}(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e}\ cbvr.r$$

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.

Unspecified Evaluation Order. The specification of Scheme, when translated into our setting is more difficult to model accurately. There are two conditions:

(1) In any application $\texttt{apply}(e_1, e_2)$, either or argument may be evaluated first.

(2) There can be no interleaving of the evaluation of the two arguments. In other words, the constituent we pick to evaluate first must be completely evaluated before picking the other.

As discussed before, such an underspecification has obvious disadvantages with respect to portability, since the code exhibits spurious non-determinism. While modelling part (1) is quite straightforward by simply including the left and right search rules, part (2) does not fit into the form of the rules that

we have specified so far. It seems that one would need either an auxiliary judgment or some auxiliary abstract syntax constructors. We show here the latter. We introduce three forms of application: uncommitted `apply`, left-to-right $\texttt{apply}_1$ and right-to-left $\texttt{apply}_2$. The first two rules commit to the choice between the two constructs.

$$\frac{}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}_1(e_1, e_2)} \; cbvs.dl$$

$$\frac{}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}_2(e_1, e_2)} \; cbvs.dr$$

The second and third set of rules step according to the left-to-right order for $\texttt{apply}_1$ and according to the right-to-left order for $\texttt{apply}_2$.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}_1(e_1, e_2) \mapsto \texttt{apply}_1(e_1', e_2)} \; cbvs.l1$$

$$\frac{v_1 \text{ value} \quad e_2 \mapsto e_2'}{\texttt{apply}_1(v_1, e_2) \mapsto \texttt{apply}_1(v_1, e_2')} \; cbvs.l2$$

$$\frac{e_2 \mapsto e_2'}{\texttt{apply}_2(e_1, e_2') \mapsto \texttt{apply}_2(e_1, e_2')} \; cbvs.r1$$

$$\frac{e_1 \mapsto e_1' \quad v_2 \text{ value}}{\texttt{apply}_2(e_1, v_2) \mapsto \texttt{apply}_2(e_1', v_2)} \; cbvs.r2$$

The final set of rules carries out the identical reductions for the two committed forms of application.

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\texttt{apply}_1(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e} \; cbvs.lr$$

$$\frac{(v_1 = \texttt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\texttt{apply}_2(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e} \; cbvs.rr$$

For this to work properly we must enforce that the constructors $\texttt{apply}_1$ and $\texttt{apply}_2$ are only used internally in the semantics, but are not accessible through the concrete syntax of the language. This is because the language does not actually provide the programmer with the explicit choice: his or her program should be correct no matter which order of evaluation is applied. Nonetheless, we need to write the (obvious) typing rules for them in order to prove progress and preservation, since all the intermediate states during evaluation must be typable.

Supplementary Notes on Aggregate Data Structures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 7
Sep 17, 2002

Before we go into aggregate data structures (pairs, sums, and some recursive types), we discuss how run-time errors can be handled in a type-safe language [Ch. 9.3]. Consider extending our MinML language by a partial division operator, $\mathtt{div}(e_1, e_2)$. Besides the usual typing rules and search rules for the operational semantics, we would also have the following reduction rule:

$$\frac{(n_2 \neq 0)}{\mathtt{div}(\mathtt{num}(n_1), \mathtt{num}(n_2)) \mapsto \mathtt{num}(\lfloor n_1/n_2 \rfloor)}$$

The condition $n_2 \neq 0$ means that there is no rule for $\mathtt{div}(\mathtt{num}(n), \mathtt{num}(0))$ and evaluation gets stuck. Progress would be violated.

We can restore an amended progress theorem if we introduce a new judgment e aborts to explicitly require that run-time errors will abort the program rather than continuing in some random state. We add the rule

$$\frac{}{\mathtt{div}(\mathtt{num}(n_1), \mathtt{num}(0)) \text{ aborts}}$$

However, we are not finished, because an expression such as

$$\mathtt{plus}(\mathtt{div}(\mathtt{num}(3), \mathtt{num}(0)), \mathtt{num}(2))$$

must also abort, but we have no rule that allows us to conclude this. So in addition to the search rules we have "abort propagation" rules that propagate run-time errors up to the overall program we are trying to evaluate.

We show the two rules for application as an example; similar rules are necessary for all search rules to account for a possible abort.

$$\frac{e_1 \text{ aborts}}{\texttt{apply}(e_1, e_2) \text{ aborts}}$$

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

Now we can refine the statements progress and determinism to account for the new judgment. Note that preservation does not change, because it only has to account for a successful computation step.

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, or
   (iii) e aborts.
3. (Determinism) If $\cdot \vdash e : \tau$ then exactly one of
   (i) $e \mapsto e'$ for some unique e', or
   (ii) e value, or
   (iii) e aborts.

We do not give her a proof of these properties, nor do we discuss how the language might be extended with a `try`...`handle`...`end` construct in order to catch error conditions.

Now we come to various language extensions which make MinML a more realistic language without changing its basic character.

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`

[1] See Assignment 3

constructor, we can extract the first and second component of a pair.[2]

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

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

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

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

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 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.

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

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

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

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

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

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

[2] An alternative treatment is given in [Ch. 19.1], where the destructor provides access to both components of a pair simultaneously. Also, the unit type comes with a corresponding `check` construct.

Unit Type. 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 in the next section on sums.

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

The value inversion property is also simple.

If $\cdot \vdash v : \texttt{unit}$ then $v = \langle\rangle$.

Sums. Unions, as one might now 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.[3]

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

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

[3]Strictly speaking, some of this information is redundant, but it is easier read if we are fully explicit here.

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.

$$\frac{e_1 \text{ value}}{\texttt{inl}(\tau_1, \tau_2, e_1) \text{ value}} \qquad \frac{e_2 \text{ value}}{\texttt{inr}(\tau_1, \tau_2, e_2) \text{ value}}$$

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

$$\frac{v_1 \text{ value}}{\texttt{case}(\texttt{inl}(\tau_1, \tau_2, v_1), x_1.e_1, x_2.e_2) \mapsto \{v_1/x_1\}e_1}$$

$$\frac{v_2 \text{ value}}{\texttt{case}(\texttt{inr}(\tau_1, \tau_2, v_2), x_1.e_1, x_2.e_2) \mapsto \{v_2/x_2\}e_2}$$

We also state the value inversion property.

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

Empty Type. The empty type can be thought of as a zero-ary sum. It has no values and a corresponding abort construct which should never be executable unless we add an error value to the language.

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}(\tau, e) : \tau}$$

There is only one search rule of computation, but no actual reduction rule.

$$\frac{e \mapsto e'}{\texttt{abort}(\tau, e) \mapsto \texttt{abort}(\tau, e')}$$

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.

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`(τ_1, τ_2, e_1)	`inl`(e_1) : τ_1`+`τ_2	$\mathsf{inl}_{\tau_1+\tau_2}(e_1)$
`inr`(τ_1, τ_2, e_2)	`inr`(e_2) : τ_1`+`τ_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`(τ, e)	`abort`(e) : τ	$\mathsf{abort}_\tau(e)$

Supplementary Notes on An Abstract Machine

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 8
Sep 19, 2002

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 represents 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	$::=$	$\mathtt{o}(\Box, e_2) \mid \mathtt{o}(v_1, \Box)$	primops
		$\mid$	$\mathtt{pair}(\Box, e_2) \mid \mathtt{pair}(v_1, \Box)$	pairs
		$\mid$	$\mathtt{fst}(\Box) \mid \mathtt{snd}(\Box)$	projections
		$\mid$	$\mathtt{apply}(\Box, e_2) \mid \mathtt{apply}(v_1, \Box)$	applications
		$\mid$	$\mathtt{if}(\Box, e_1, e_2)$	conditional

A hole $\Box$ 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 machines 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 $k < v$.

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

Integers.

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

Products.

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

Functions.

$$
\begin{array}{lcl}
k > \mathsf{fun}(\tau_1, \tau_2, f.x.e) & \mapsto_{\mathsf{c}} & k < \mathsf{fun}(\tau_1, \tau_2, f.x.e) \\
\\
k > \mathsf{apply}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \mathsf{apply}(\Box, e_2) > e_1 \\
k \triangleright \mathsf{apply}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k \triangleright \mathsf{apply}(v_1, \Box) > e_2 \\
k \triangleright \mathsf{apply}(v_1, \Box) < v_2 & \mapsto_{\mathsf{c}} & k > \{v_1/f\}\{v_2/x\}e \\
 & & \quad (v_1 = \mathsf{fun}(\tau_1, \tau_2, f.x.e))
\end{array}
$$

Conditionals.

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

As an example, consider the evaluation of

```
(fun f(x:int):int is x end) 0
```

We elide the types `int` in order to shorten the syntax.

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

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

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 1 (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 2 (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 3 (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 4 (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 $v = \mathtt{fun}(\tau_1, \tau_2, f.x.e)$. 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

■

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. It is difficult to explain how one might arrive at this generalization, except to say "through experience" and by analysing the failure of other attempts. 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 e.

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

Lemma 5 (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
$k > \mathtt{pair}(e_1', e_2) \quad \mapsto_{\mathsf{c}}^* \bullet > a$	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:

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

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

$k \triangleright \texttt{pair}(\Box, e_2) > v_1 \mapsto^* \bullet > a$ By value computation (i)
$k > \texttt{pair}(v_1, e_2) \mapsto k \triangleright \texttt{pair}(\Box, e_2) > v_1 \mapsto^* \bullet > a$ 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:

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

$k < v_2 \mapsto_{\mathsf{c}}^* \bullet > a$ Assumption

$k > \texttt{snd}(\texttt{pair}(v_1, v_2))$
$\mapsto_{\mathsf{c}} k \triangleright \texttt{snd}(\Box) > \texttt{pair}(v_1, v_2)$ By rule
$\mapsto_{\mathsf{c}}^* k \triangleright \texttt{snd}(\Box) < \texttt{pair}(v_1, v_2)$ 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 correctness theorem.

Theorem 6 (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 Exceptions

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 9
September 25, 2002

In this lecture we first give an implementation of the C-machine for the fragment containing integers, booleans, and functions using higher-order functions. We then discuss exceptions as an extension of the C-machine [Ch. 13]

The implementation of the C-machine is to represent the stack as a *continuation* that encapsulates the rest of the computation to be performed.[1]

First, in our implementation, both expressions and value have type `exp`. We nonetheless use different names to track our intuition, even though the type system of ML does not help use verify the correctness of this intuition.

```
type value = exp
e : exp
v : value
```

Next, the stack k is represented by an ML function

```
k : value -> value
```

Applying this function to a value v will carry out the rest of the computation of the machine, returning the final answer. Finally, we have two functions

[1] We give some code excerpts here; the full code can be found at `http://www.cs.cmu.edu/~fp/courses/312/code/09-exceptions/`.

```
eval : exp -> (value -> value) -> value
return : value -> (value -> value) -> value
```

satisfying the specification:

(i) `eval` $e\ \ k\ \Downarrow\ a$ iff $k > e \mapsto_{\mathsf{c}}^{*} \bullet < a$

(ii) `return` $v\ \ k\ \Downarrow\ a$ iff $k < v \mapsto_{\mathsf{c}}^{*} \bullet < a$

In order to implement stacks as ML functions, it is useful to introduce some new auxiliary functions to represent the frames. We give in the table below the association between forms of the stack and the corresponding ML function. We omit only the case for primops which requires a simple treatment of lists.

$k \triangleright$ `if`$(\Box, e_2, e_3)$	`(fn v1 => ifFrame (v1, e2, e3) k)`
$k \triangleright$ `apply`$(\Box, e_2)$	`(fn v1 => applyFrame1 (v1, e2) k)`
$k \triangleright$ `apply`$(v_1, \Box)$	`(fn v2 => applyFrame2 (v1, v2) k)`
$k \triangleright$ `let`$(\Box, x.e_2)$	`(fn v1 => letFrame (v1, ((), e2)) k)`
$\bullet$	`(fn v => v)`

The case of the empty stack corresponds to the initial continuation, which simply returns the value passed to it as the result of the overal computation

Now we can piece together the whole code elegantly, as advertised. We have elided only the case for primitive operations, which can be found with the complete code at the address given above.

```
fun eval (v as Int _) k = return v k
    (* elided primops *)
  | eval (v as Bool _) k = return v k
  | eval (If(e1, e2, e3)) k =
      eval e1 (fn v1 => ifFrame (v1, e2, e3) k)
  | eval (v as Fun _) k = return v k
  | eval (Apply(e1, e2)) k =
      eval e1 (fn v1 => applyFrame1 (v1, e2) k)
  | eval (Let(e1, ((), e2))) k =
      eval e1 (fn v1 => letFrame (v1, ((), e2)) k)
  (* eval (Var _) k impossible by MinML typing *)
and ifFrame (Bool(true), e2, e3) k = eval e2 k
  | ifFrame (Bool(false), e2, e3) k = eval e3 k
  (* other expressions impossible by MinML typing *)
and applyFrame1 (v1, e2) k =
       eval e2 (fn v2 => applyFrame2 (v1, v2) k)
and applyFrame2 (v1 as Fun(_, _, ((), (), e1')), v2) k =
       eval (Subst.subst (v1, 2, Subst.subst (v2, 1, e1'))) k
  (* other expressions impossible by MinML typing *)
and letFrame (v1, ((), e2)) k = eval (Subst.subst (v1, 1, e2)) k
and return v k = k v
```

The overall evaluation just starts with the initial continuation which corresponds to the empty stack.

```
fun evaluate e = eval e (fn v => v)
```

This style of writing an interpreter is also refered to as *continuation-passing style*. It is quite flexible and elegant, and will be exercised in Assignment 4.

Next we come to exceptions. We introduce a new form of state

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

which signals that we are propagation 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. For more complex variations of exceptions, see [Ch. 13] and Assignment 4.

We have two new forms of expressions `fail`(τ) (with concrete syntax

$\texttt{fail}[\tau]$[2] and $\texttt{try}(e_1, e_2)$ (with concrete syntax $\texttt{try}\, e_1\, \texttt{ow}\, e_2$). Informally, $\texttt{try}(e_1, e_2)$ 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.

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

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:

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

The new judgment for typing states depends on a typing for stacks. A stack is characterized by the type of the argument it expects and the type of the final answer it returns. We write ρ for the type of the final answer. Note that during the whole computation of a machine, this never changes. The new judgments are

$s\ \mathsf{OK}_\rho$	state s is well-formed with final answer type ρ
$k\ : \tau\ \mathsf{stack}_\rho$	stack k accepts a value of type τ and returns a final answer of type ρ

Since ρ never changes for any run of the machine, we omit the subcript in some of the rules below. However, keep in mind that it is implicitly present. Note also that judgments on states and stacks do not need to be hypothetical judgments, since they never contain free variables. First, the rules for states which ensure that the type expected by a stack matches the type of the expression to be evaluated, or value being returned.

[2] The type is written here in order to preserve the property that every well-typed expression has a unique type.

$$\frac{k \ : \tau \ \mathsf{stack}_\rho \quad \cdot \vdash e : \tau}{k > e \ \ \mathsf{OK}_\rho}$$

$$\frac{k \ : \tau \ \mathsf{stack}_\rho \quad \cdot \vdash v : \tau \quad v \ \ \mathsf{value}}{k < v \ \ \mathsf{OK}_\rho}$$

$$\frac{k \ : \tau \ \mathsf{stack}_\rho}{k \ll \mathsf{fail} \ \ \mathsf{OK}_\rho}$$

The rules for stacks are straightforward, given a few examples below.

$$\frac{}{\bullet \ : \rho \ \mathsf{stack}_\rho}$$

$$\frac{k \ : \tau_1 \ \mathsf{stack} \quad \cdot \vdash e_2 : \tau_2}{k \triangleright \mathtt{apply}(\Box, e_2) \ : \tau_2 \rightarrow \tau_1 \ \mathsf{stack}}$$

$$\frac{k \ : \tau_1 \ \mathsf{stack} \quad \cdot \vdash v_1 : \tau_2 \rightarrow \tau_1 \quad v_1 \ \mathsf{value}}{k \triangleright \mathtt{apply}(v_1, \Box) \ : \tau_2 \ \mathsf{stack}}$$

$$\frac{k \ : \tau \ \mathsf{stack} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{k \triangleright \mathtt{if}(\Box, e_2, e_3) \ : \mathtt{bool} \ \mathsf{stack}}$$

$$\frac{k \ : \tau \ \mathsf{stack} \quad \cdot \vdash e_2 : \tau}{k \triangleright \mathtt{try}(\Box, e_2) \ : \tau \ \mathsf{stack}}$$

$$\frac{k \ : \tau_2 \ \mathsf{stack} \quad x{:}\tau_1 \vdash e_2 : \tau_2}{k \triangleright \mathtt{let}(\Box, x.e_2) \ : \tau_1 \ \mathsf{stack}}$$

We can now state (without proof) the preservation and progress properties. The proofs follow previous patterns (see [Ch. 13]) and Lecture 5 on *Type Safety*.

1. (Preservation) If $s \ \ \mathsf{OK}_\rho$ and $s \mapsto s'$ then $s' \ \ \mathsf{OK}_\rho$.

2. (Progress) If $s \ \ \mathsf{OK}_\rho$ then either

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

   (ii) $s = \bullet < v$ with v value, or

   (iii) $s = \bullet \ll \mathsf{fail}$.

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, this 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.

In our simple language, the handler stack h contains only frames $\mathtt{ow}(k, e_2)$ while the control stack contains the usual frames, and $\mathtt{try}(\Box)$ (the "otherwise" clause has moved to the handler stack). All the usual rules are augmented to carry a control stack and a handler stack, and leave the handler unchanged.

$$
\begin{array}{lcl}
(h, k) > \mathtt{apply}(e_1, e_2) & \mapsto_{\mathsf{c}} & (h, k \triangleright \mathtt{apply}(\Box, e_2)) > e_1 \\
\ldots & & \\
(h, k) > \mathtt{try}(e_1, e_2) & \mapsto_{\mathsf{c}} & (h \triangleright \mathtt{ow}(k, e_2), k \triangleright \mathtt{try}(\Box)) > e_1 \\
(h \triangleright \mathtt{ow}(k', e_2), k \triangleright \mathtt{try}(\Box)) < v_1 & \mapsto_{\mathsf{c}} & (h, k) < v_1 \\
(h \triangleright \mathtt{ow}(k', e_2), k) > \mathtt{fail}(\tau) & \mapsto_{\mathsf{c}} & (h, k') > e_2
\end{array}
$$

Note that we do not unwind the control stack explicilty, but jump directly to the handler when an exception is raised. This handler must story a copy of the control stack in effect at the time the `try` expression was executed. Fortunately, this can be implemented without the apparent copying of the stack in the rule for `try`, because we can just keep a pointer to the right frame in the control stack [Ch. 13].

Note also in case of a regular return for the subject of a `try` expression, we need to pop the corresponding handler off the handler stack.

Supplementary Notes on Continuations

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 10
September 26, 2002

In this lecture we introduce *continuations*, an advanced control construct available in some functional languages [Ch. 12]. Most notably, they 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.[1]

`letcc` x `in` e `end`	`letcc`$(\tau, x.e)$
`throw` e_1 `to` e_2	`throw`(τ, e_1, e_2)

In brief, `letcc` x `in` e `end` captures the stack (= continuation) k in effect at the time the `letcc` 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 passed 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 an an alternative to exceptions, another is to implement co-routines or thread. Another use is to affect backtracking.

As a starting example we consider simple arithmetic expressions.

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

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

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.

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

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.

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

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

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.

$$\frac{k : \tau\ \mathsf{stack}}{\Gamma \vdash \texttt{cont}(k) : \tau\ \mathsf{cont}}$$

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 (letcc k1 in ... end)) 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.

```
letcc r
in
   throw (F (letcc k1 in throw k1 to r end)) to K
end
```

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
  letcc r
  in
     throw (f (letcc k1 in throw k1 to r end)) to k
  end
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).

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

At this point, we define

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

and continue

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

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[2] 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
`letcc` x `in` e `end`	`letcc`$(\tau, x.e)$	`callcc (fn x => e)`
`throw` e_1 `to` e_2	`throw`(τ, 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.

[2] `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
October 1, 2002

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 `fun f(x) is x end`) 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. 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.

```
fun map (f:int -> bool):int list -> bool list is
  fun _ (l:int list):bool list is
    case l
      of nil => nil[bool]
       | cons(x,l') => cons(f(x),map f l')
  end
end
```

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 objected 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 `Fun` t `in` e `end` 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`$(t.\tau)$	$\forall t.\tau$
`Fun` t `in` e `end`	`Fun`$(t.e)$	$\Lambda t.e$
e `[`τ`]`	`Inst`(e, τ)	$e[\tau]$
`fun` f `(`x`:`τ_1`):`τ_2 `is` e `end`	`fun`$(\tau_1, \tau_2, f.x.e)$	$\mu f{:}\tau_1 \to \tau_2.\ \lambda x{:}\tau_1.\ e$
`fun _(`x`:`τ_1`):_ is` e `end`	`fun`$(\tau_1, \_, \_.x.e)$	$\lambda x{:}\tau_1.\ e$
`fn` x`:`τ_1 `=> e`		

Using this notation, we can rewrite the example above.

```
Fun t in Fun s in
fun map (f:t -> s):t list -> s list is
  fun _ (l:t list):s list is
    case l
      of nil => nil[s]
       | cons(x,l') => cons(f(x),map f l')
  end
end
end end
```

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.

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

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

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.

$$\begin{array}{rcl} t \text{ type}, f{:}\texttt{arrow}(t,t), x{:}t & \vdash & x : t \\ t \text{ type} & \vdash & \texttt{fun}(t, t, f.x.x) : \texttt{arrow}(t,t) \\ \cdot & \vdash & \texttt{Fun}(t.\texttt{fun}(t, t, f.x.x)) : \texttt{All}(t.\texttt{arrow}(t,t)) \end{array}$$

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.

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

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

$$\begin{array}{rcl} \cdot & \vdash & id : \forall t.t \to t \\ \cdot & \vdash & id\,[\texttt{int}] : \texttt{int} \to \texttt{int} \\ \cdot & \not\vdash & id\,[\texttt{int}]\,\texttt{true} : \texttt{int} \end{array}$$

Using mathematical notation and the short form for a non-recursive function:

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

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 substitution 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 $\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$ 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:

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

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

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* $v\ \ \mathsf{value}$ *then* $v = \mathsf{Fun}(t.e')$ *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) $e\ \ \mathsf{value}$, *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 technique that we may not get to 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 may return to the issue of type inference later in this course.

Supplementary Notes on Data Abstraction

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 12
October 3, 2002

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. 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 `int list` $\times$ `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 $\Gamma \vdash \sigma$ 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

$$\texttt{Fun}(a.\texttt{pack}(a\ \texttt{list}, \langle \texttt{Inst}(\texttt{nil}, a), \ldots \rangle))$$

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 `qq` with some additional operations that we leave unspecified here. Then the type would have the form

$$\exists q.\exists qq.q \times qq \times \cdots.$$

The implementation would provide definitions for both q and qq, 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]

$$\texttt{pack}(\texttt{int}, \texttt{pair}(\lambda x.x + 1, \lambda x.x - 1)).$$

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] Recall the abbreviation $\lambda x.e$ for a function $\texttt{fun}(\_, \_, f.x.e)$ where e does not depend on f, that is, does not make a recursive call.

Supplementary Notes on Recursive Types

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 13
October 8, 2002

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 create 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.

Supplementary Notes on Mutable Storage

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 14
October 10, 2002

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* [Ch. 14].

We will look at mutable storage from two different points of view: one, where essentially all of MinML becomes an imperative language (this lecture), 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 $\tau\,\mathsf{ref}$ 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.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Since locations can be mentioned anywhere in a program, we 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 ML.

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \mathsf{ref}(e) : \tau\,\mathsf{ref}} \qquad \frac{\Lambda; \Gamma \vdash e_1 : \tau\,\mathsf{ref} \quad \Lambda; \Gamma \vdash e_2 : \tau}{\Lambda; \Gamma \vdash e_1 := e_2 : \tau}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau\,\mathsf{ref}}{\Lambda; \Gamma \vdash !e : \tau} \qquad \frac{l{:}\tau \text{ in } \Lambda}{\Lambda; \Gamma \vdash l : \tau\,\mathsf{ref}}$$

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.

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.

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, \mathtt{apply}(e_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(e_1', e_2)\rangle}$$

$$\frac{v_1 \text{ value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, \mathtt{apply}(v_1, e_2)\rangle \mapsto \langle M', \mathtt{apply}(v_1, e_2')\rangle}$$

$$\frac{(v_1 = \mathtt{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\langle M, \mathtt{apply}(v_1, v_2)\rangle \mapsto \langle M, \{v_1/f\}\{v_2/x\}e\rangle}$$

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.

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M', \mathsf{ref}(e')\rangle} \qquad \frac{v \ \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), l\rangle} \qquad \frac{}{l \ \ \mathsf{value}}$$

$$\frac{\langle M, e_1\rangle \mapsto \langle M', e_1'\rangle}{\langle M, e_1 := e_2\rangle \mapsto \langle M', e_1' := e_2\rangle} \qquad \frac{v_1 \ \ \mathsf{value} \quad \langle M, e_2\rangle \mapsto \langle M', e_2'\rangle}{\langle M, v_1 := e_2\rangle \mapsto \langle M', v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2)}{\langle M, l := v_2\rangle \mapsto \langle M', v_2\rangle}$$

$$\frac{\langle M, e\rangle \mapsto \langle M', e'\rangle}{\langle M, !e\rangle \mapsto \langle M', !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !l\rangle \mapsto \langle M, v\rangle}$$

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 refer other cells which can in turn refer back to the first cell. In other words, the pointer structure 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.

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 too familiar with.

Supplementary Notes on Monads

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 15
October 15, 2002

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 Assignment 5 you are asked to instantiate the monadic framework instead by defining a simple semantics of input and output.

In the generic framework, we extend MinML by adding a new syntactic category of *monadic expressions*, denoted by m. Correspondingly, there is a new typing judgment

$$\Gamma \vdash m \div \tau$$

[1] Lazy here means call-by-name with memoization of the suspension.

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 has 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 $\mathsf{val}(m)$.

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{val}(m) : \bigcirc\tau} \qquad \frac{}{\mathsf{val}(m)\ \ \mathsf{value}}$$

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 with an explicit sequence of monadic expressions! This yields the following construct

$$\frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{let\ val}\ x = e\ \mathsf{in}\ m\ \mathsf{end} \div \sigma}$$

Note that m and $\mathsf{let\ val}\ x = e\ \mathsf{in}\ m\ \mathsf{end}$ 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 $\mathsf{val}(m')$.

(2) We execute the monadic expression m', which will have some effects but also return a value in the form $[v]$.

(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. For this we introduce the concept of *worlds* w that encapsulate all state that may be changed by an effect. In the case of the storage monads, this will be the memory M. In the case of the I/O monad, this will be input and output streams.

The judgment for executing monadic expressions then has the form

$$\langle w, m\rangle \mapsto \langle w', m'\rangle$$

where the world changes from w to w' and the expression steps from m to m'. According to the considerations above, we obtain the following rules.

$$\frac{e \mapsto e'}{\langle w, [e]\rangle \mapsto \langle w, [e']\rangle}$$

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.

$$\frac{e \mapsto e'}{\langle w, \mathsf{let\ val}\, x = e\, \mathsf{in}\, m\, \mathsf{end}\rangle \mapsto \langle w, \mathsf{let\ val}\, x = e'\, \mathsf{in}\, m\, \mathsf{end}\rangle}$$

$$\frac{\langle w, m_1\rangle \mapsto \langle w', m_1'\rangle}{\langle w, \mathsf{let\ val}\, x = \mathsf{val}(m_1)\, \mathsf{in}\, m\, \mathsf{end}\rangle \mapsto \langle w', \mathsf{let\ val}\, x = \mathsf{val}(m_1')\, \mathsf{in}\, m\, \mathsf{end}\rangle}$$

$$\frac{}{\langle w, \mathsf{let\ val}\, x = \mathsf{val}([v])\, \mathsf{in}\, m\, \mathsf{end}\rangle \mapsto \langle w, \{v/x\}m\rangle}$$

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.

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \mathsf{ref}(e) \div \tau\ \mathsf{ref}} \qquad \frac{\Lambda;\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Lambda;\Gamma \vdash e_2 : \tau}{\Lambda;\Gamma \vdash e_1 := e_2 \div \tau}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\ \mathsf{ref}}{\Lambda;\Gamma \vdash !e \div \tau} \qquad \frac{l{:}\tau\ \text{in}\ \Lambda}{\Lambda;\Gamma \vdash l : \tau\ \mathsf{ref}}$$

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 letval form.

In order to describe the operational semantics we need to make the worlds explicit. In this case a world consists simply of the current memory M. Recall the form of stores.

$$\textit{Stores} \qquad M ::= \cdot \mid M, l{=}v$$

Now the additional rules for new expressions are analogous to those from the previous lecture.

$$\frac{e \mapsto e'}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M, \mathsf{ref}(e')\rangle} \qquad \frac{v\ \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), [l]\rangle} \qquad \frac{}{l\ \ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\langle M, e_1 := e_2\rangle \mapsto \langle M, e_1' := e_2\rangle} \qquad \frac{v_1\ \ \mathsf{value} \quad e_2 \mapsto e_2'}{\langle M, v_1 := e_2\rangle \mapsto \langle M, v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2)}{\langle M, l := v_2\rangle \mapsto \langle M', [v_2]\rangle}$$

$$\frac{e \mapsto e'}{\langle M, !e\rangle \mapsto \langle M, !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !l\rangle \mapsto \langle M, [v]\rangle}$$

The progress and type preservation theorems now need to be extended to cover both pure and monadic expressions. We also seen to verify that a store satisfies a store typing. Recall the rules for the judgment $\Lambda_0; \cdot \vdash M : \Lambda$.

$$\frac{}{\Lambda_0; \cdot \vdash (\cdot) : (\cdot)} \qquad \frac{\Lambda_0; \cdot \vdash M : \Lambda \quad \Lambda_0; \cdot \vdash v : \tau \quad v\ \ \mathsf{value}}{\Lambda_0; \cdot \vdash (M, l{=}v) : (\Lambda, l{:}\tau)}$$

We can now formulate the appropriate generalizations of type preservation and progress. 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.

Theorem 1 (Type Preservation)
(1) If $\Lambda; \cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\Lambda; \cdot \vdash e' : \tau$.

(2) If $\Lambda; \cdot \vdash m \div \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *and* $\langle M, m\rangle \mapsto \langle M', m'\rangle$ *then for some* $\Lambda' \geq \Lambda$ *and memory* M' *we have* $\Lambda'; \cdot \vdash m' \div \tau$ *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)
(1) If $\Lambda; \cdot \vdash e : \tau$ *then either*

(i) e value, *or*

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

(2) If $\Lambda; \cdot \vdash m \div \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *then either*

(i) $m = [v]$ *for some* v value, *or*

(ii) $\langle M, m\rangle \mapsto \langle M', m'\rangle$ *for some* M' *and* m'.

Proof: By induction on the derivation of the typing judgment, analyzing all possible cases. ■

As an example consider a function inc : int ref $\rightarrow \bigcirc$int which takes a location as an argument, increments it, and returns the incremented value. The return type has to be protected by the monadic type, since the function has an effect.

$$
\begin{array}{lcl}
inc & : & \mathsf{int\ ref} \rightarrow \bigcirc\mathsf{int} \\
 & = & \lambda r.\mathsf{val}(\mathsf{let\ val}(x_1) = \mathsf{val}(!r)\ \mathsf{in} \\
 & & \qquad \mathsf{let\ val}(x_2) = \mathsf{val}(r := x_1 + 1)\ \mathsf{in} \\
 & & \qquad [x_2]\mathsf{end\ end})
\end{array}
$$

Several things to note about this definition. When inc is called on a location, it *returns* an effectful computation, but it does not carry it out (val(m) quotes m). Secondly, the first let expression is necessary, because $r := !r + 1$

incorrectly uses the monadic expression $!r$ in a place where a pure expression is expected. The uses of $\mathsf{val}(m)$ on the right-hand side of the lets can be avoid by introducing appropriate definitions such as

$$\begin{array}{lcl} \mathsf{new} & : & \forall t.t \to \bigcirc(t\,\mathsf{ref}) \\ & = & \Lambda t.\lambda x.\mathsf{val}(\mathsf{ref}(x)) \\ \mathsf{get} & : & \forall t.t\,\mathsf{ref} \to \bigcirc t \\ & = & \Lambda t.\lambda r.\mathsf{val}(!r) \\ \mathsf{set} & : & \forall t.t\,\mathsf{ref} \to t \to \bigcirc t \\ & = & \Lambda t.\lambda r.\lambda x.\mathsf{val}(r := x) \end{array}$$

Furthermore, the Haskell language creates some syntactic sugar that makes it easier to write a sequence of let val forms in a row.

In order to create a cell, initialize it with 0, increment it once an return cell's contents we can write the following monadic expression at the top level.

$$\begin{array}{l} \mathsf{let}\ \mathsf{val}(x_1) = new(0)\ \mathsf{in} \\ \mathsf{let}\ \mathsf{val}(x_2) = inc(x_1)\ \mathsf{in} \\ \mathsf{let}\ \mathsf{val}(x_3) = get(x_1)\ \mathsf{in} \\ [x_3]\ \mathsf{end}\,\mathsf{end}\,\mathsf{end} \\ \div\mathsf{int} \end{array}$$

When started in the empty memory, the above monadic expression executes and evaluates to $\langle (l{=}1), 1\rangle$. 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.

Supplementary Notes on Subtyping

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 16
October 22, 2002

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 programs. Finally, it is absolutely fundamental in object-oriented programming where the notion of a 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 careful justifies 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. We show the special 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 $n = n'$ or e and e' both do not terminate.
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:

$$\dfrac{\dfrac{\overline{\vdash 2 : \mathsf{int}} \quad \overline{\mathsf{int} \leq \mathsf{float}}}{\vdash 2 : \mathsf{float}} \qquad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash 2 +. 3.0 : \mathsf{float}}$$

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

$$\dfrac{\dfrac{\overline{\vdash 2 : \mathsf{int}} \quad \overline{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\vdash \mathsf{itof}(2) : \mathsf{float}} \qquad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash \mathsf{itof}(2) +. 3.0 : \mathsf{float}}$$

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. For example, we cold have

$$\dfrac{\dfrac{}{\vdash 2 : \mathsf{int}} \quad \dfrac{\overline{\lambda x.x : \mathsf{int} \leq \mathsf{int}} \quad \overline{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\lambda y.\mathsf{itof}((\lambda x.x)(y)) : \mathsf{int} \leq \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) : \mathsf{float}} \qquad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) +. 3.0 : \mathsf{float}}$$

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 component.

Mutable reference can be neither covariant nor contravariant. As simple counterexamples, consider the following pieces of code.

The first one assumes that $\tau\ \mathsf{ref} \subset \sigma\ \mathsf{ref}$ 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, $\tau\ \mathsf{ref} \leq \sigma\ \mathsf{ref}$ 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*.

$$\frac{}{\tau\ \mathsf{ref} \leq \tau\ \mathsf{ref}}$$

Of course, this is already entailed by the reflexivity rule. 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. Source 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 the the non-variance of reference 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.

Supplementary Notes
Lecture 17: Bidirectional Typing

15-312: Foundations of Programming Languages
Joshua Dunfield (joshuad@cs.cmu.edu)

24 October 2002

1 Type Inference: The Good, The Bad

So far (as formulated in Assignments 2 and 4, for instance), the typing problem has always been:

> Given a context Γ and term e, produce its type τ (or fail if it has no type).

This is the same problem as type inference in core SML (SML without modules). It's possible to annotate any SML expression with a type; this is often highly desirable, given SML/NJ's suboptimal type error reporting, since it tends to produce type error messages in which the claimed error site actually *is* the error site. But it is *never* necessary (again, *without* modules—we've already seen how existential types make type inference impossible). We haven't really tried to do this in MinML; while the form of the problem is the same, we have always required certain types to be explicitly annotated, in particular, on function declarations. As we expanded the language, this became increasingly annoying; the type annotation on **raise**, for example, seems particularly gratuitous.

It's well understood how to do full type inference, SML-style: generate constraints and unify the variables. Why haven't we done this, since it would allow us to get rid of the type annotations on **raise** and so forth? There are two reasons:

1. It's somewhat complicated. It would probably be a full programming assignment, and there are more interesting things to do.

2. After a while, it stops working.

What do I mean by that? If you add existentials to your language, you can't infer all types. If you add sums, you can't infer all types—what is the type of

$$\texttt{inl}(5)$$

? If you add something more exotic, like intersection types or refinement types, you can't infer all types. Most relevantly, **you can't do subtyping**. Consider the subsumption typing rule

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e : \tau} \text{ (Sub)}$$

What does this say? To infer type τ for e, we must infer type σ for e and show $\sigma \leq \tau$. Right away there's a problem: there's nothing to keep us from recursing forever. But hey, infinite recursion is silly. We can just agree to not apply the subsumption rule twice in a row. (Why is this complete?)

Unfortunately, that was the least of our problems. We have $e : \sigma$. But we have to show $\sigma \leq \tau$. What is τ? We have no idea. We have to *guess* a τ that will make the rest of the typing derivation work.[1]

Life would be so much easier if we were *given* the type, instead of having to infer it.

2 No Type Inference: The Ugly

Instead of inferring types, let's check them. The problem of typing becomes

> Given a context Γ, a term e, and a type τ, return **true** iff e checks against τ.

Now everything is very easy. There's just one little problem: if we do this everywhere, we have to write so many type annotations that the language becomes unusable.

```
(2 : int) + (2 : int) : int
```

3 Some Type Inference

In practice, languages use some mixture of inferred types and checked types. C[2] and Java are examples: types have to be given with all functions

[1] This is similar to the problem with implementing the transitivity rule, discussed in recitation.

[2] Of course, C's type system is almost meaningless, but that's beside the point.

and variable declarations, but not for things like `2 + 2`. SML is another example: inference suffices for the core language but not the module language, since modules correspond to existential types. Saying that structure FOO ascribes to signature Foo amounts to writing an annotation on an existential type.

Likewise, intersection types $\sigma \,\&\, \tau$ can't be inferred because in the rule

$$\frac{\Gamma \vdash v : \sigma \qquad \Gamma \vdash v : \tau}{\Gamma \vdash v : \sigma \,\&\, \tau} \text{ (\&Intro)}$$

one has to guess both σ and τ (and in fact there are an infinite number of intersection types for any well-typed term: $\sigma \,\&\, \sigma$, $(\sigma \,\&\, \sigma) \,\&\, \sigma$, ...).

We would like a system that is

1. Practical (not just decidable, but efficient too);
2. Usable (needing only a reasonable number of type annotations, easy to understand, predictable, good error reporting, ...?);
3. Aesthetically pleasing.

4 Bidirectional Typing

The idea is to have both inference and checking judgments in the same system. In the inference judgment

$$\Gamma \vdash e \uparrow \tau$$

one is given Γ and e and must produce a τ (or fail). In the checking judgment

$$\Gamma \vdash e \downarrow \tau$$

one is given everything—Γ, e, and τ—and must simply return true iff the judgment is derivable.

We can move between the two judgments as follows. First we introduce a new syntactic form, the type annotation, written

$$e : \tau \qquad \texttt{Anno}(e, \tau)$$

If we need to infer a type for a term, but no type can be inferred (for example, if the term is a `pack`), the user has to give a type annotation. Then we can check against the annotation. The rule is

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau} \text{ (Anno)}$$

So we can move from inferring a type for an annotated term to checking the term against a type. How can we move in the other direction? If we can infer type τ for e, e certainly should check against τ.

$$\frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash e \downarrow \tau} \text{ (AlmostSub)}$$

However, the subsumption rule subsumes[3] this rule. The bidirectional subsumption rule is

$$\frac{\Gamma \vdash e \uparrow \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e \downarrow \tau} \text{ (Sub)}$$

Now (AlmostSub) is derivable from (Sub) using reflexivity of subtyping.

These are the only rules where we move between inference and checking of the entire term e; in the other rules, we will variously check or infer *sub*terms of the term e whose type is being inferred or checked against.

In general, for each syntactic form in MinML, we have a rule concluding $\ldots \uparrow \tau$ or $\ldots \downarrow \tau$. In some cases, it's easy to see which is appropriate. The types of free variables can always be found in the context Γ, so we can always infer a type for a variable, and we can always infer a type for a `num`.

$$\frac{\Gamma = \Gamma_1, x{:}\tau, \Gamma_2}{\Gamma \vdash x \uparrow \tau} \text{ (Var)} \qquad \frac{}{\Gamma \vdash \texttt{num}(\bar{k}) \uparrow \texttt{int}} \text{ (Int)}$$

To type a function (without type annotations), we might try

$$\frac{\Gamma, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \uparrow \sigma \to \tau} \text{ (bad-Fun)}$$

But this would require us to guess both σ and τ. What we want is

$$\frac{\Gamma, f{:}\sigma{\to}\tau, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \downarrow \sigma \to \tau} \text{ (Fun)}$$

(Note about new vs. old syntax for annotating functions.) The rule for lambdas is just the same but without f:

$$\frac{\Gamma, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \lambda x.\, e \downarrow \sigma \to \tau} \text{ (Lam)}$$

To type an application $e_1 e_2$:

$$\frac{\Gamma \vdash e_1 \uparrow \sigma \to \tau \qquad \Gamma \vdash e_2 \downarrow \sigma}{\Gamma \vdash e_1 e_2 \uparrow \tau} \text{ (App)}$$

Example: $(\lambda x.\, x * 2)$ `:` $\texttt{int} \to \texttt{int}$

[3] Ow.

$$\frac{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

We type the body of the λ as follows: the body is an application, so we want to use the (App) rule, but (App) is an inference judgment and we are trying to derive a checking judgment. So we use the subsumption rule.

$$\frac{\dfrac{x{:}\texttt{int} \vdash x * 2 \uparrow \qquad\qquad \leq \texttt{int}}{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

Then we use a rule for $*$ (write it!), which infers its result and checks its arguments, analogous to (App).

$$\frac{\dfrac{\dfrac{x{:}\texttt{int} \vdash x{:}\texttt{int} \quad x{:}\texttt{int} \vdash 2{:}\texttt{int}}{x{:}\texttt{int} \vdash x * 2 \uparrow \texttt{int}} \qquad \texttt{int} \leq \texttt{int}}{x{:}\texttt{int} \vdash x * 2 \downarrow \texttt{int}}}{\vdash \lambda x.\, x * 2 \downarrow \texttt{int} \to \texttt{int}}$$

Example:

$$\texttt{fun}\ \mathit{mapdouble}(\ell)\ \texttt{is}\ \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell\ \texttt{end} : \texttt{intlist} \to \texttt{intlist}$$

Let $\Gamma = \mathit{intmap} : (\texttt{int} \to \texttt{int}) \to \texttt{intlist} \to \texttt{intlist},\ \ \ell : \texttt{intlist}$.

$$\frac{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

The next step is to use (App). The function is $\mathit{intmap}\ (\lambda x.\, x * 2)$ and we're applying it to ℓ, so we need to infer a type for $\mathit{intmap}\ (\lambda x.\, x * 2)$.

$$\frac{\dfrac{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

But $\mathit{intmap}\ (\lambda x.\, x * 2)$ is also an application, so we use (App) again.

$$\frac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \qquad \Gamma \vdash (\lambda x.\, x * 2) \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow} \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad\qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

Now we have to infer a type for *intmap*, but we can always infer a type for a variable by using (Var). The first argument to *intmap* has type $\texttt{int} \to \texttt{int}$, so we must check $\lambda x.\, x * 2$ against $\texttt{int} \to \texttt{int}$—which we just did in the preceding example.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \ldots \qquad \Gamma \vdash (\lambda x.\, x * 2) \downarrow \texttt{int} \to \texttt{int}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \texttt{intlist} \to \texttt{intlist}} \qquad \Gamma \vdash \ell \downarrow}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow} \qquad \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathit{intmap} \uparrow \ldots \quad \Gamma \vdash (\lambda x.\, x * 2) \downarrow \texttt{int} \to \texttt{int}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2) \uparrow \texttt{intlist} \to \texttt{intlist}} \quad \Gamma \vdash \ell \downarrow \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \uparrow \texttt{intlist}} \qquad \texttt{intlist} \leq \texttt{intlist}}{\Gamma \vdash \mathit{intmap}\ (\lambda x.\, x * 2)\ \ell \downarrow \texttt{intlist}}$$

4.1 Let

$$\dfrac{\Gamma \vdash e_1 \uparrow \sigma \qquad \Gamma, x{:}\sigma \vdash e_2 \downarrow \tau}{\Gamma \vdash \texttt{let}(e_1, x.e_2) \downarrow \tau}\ \text{(Let)}$$

4.2 Sums

So it seems to work nicely for functions. Let's look at sums, which were annoying without bidirectional typing because (for example) $\texttt{inl}(5)$ doesn't have a unique type. Since it doesn't have a unique type, we need to check it against a type rather than try to infer a type.

$$\dfrac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \texttt{inl}(e) \downarrow \tau_1 + \tau_2}\ \text{(Inl)} \qquad \dfrac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{inr}(e) \downarrow \tau_1 + \tau_2}\ \text{(Inr)}$$

$$\dfrac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 \downarrow \sigma \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{case}(e, x_1.e_1, x_2.e_2) \downarrow \sigma}\ \text{(Case)}$$

A peculiarity of bidirectional typing is that it doesn't work for many contrived programs. For example, $\texttt{case}(\texttt{inl}(5), x_1.0, x_2.1)$ `:` $\texttt{int}$ is not well-typed unless we annotate $\texttt{inl}(5)$. But no real program would create a sum and immediately take it apart. In practice, one almost always does a case on a variable, or on some function application—in which cases we can infer the type, and need no annotation.

Moreover, whenever an expression appears as the body of a function, we check it against the (usually annotated) result type of the function. So we can return an injection from a function with no additional type annotations, beyond the type annotation for the function. And, as we saw in the *intmap* example, sometimes we don't even need to annotate the function.

Before looking at how bidirectional typing behaves with other types, let's consider the formal properties of the system.

4.3 Soundness and Completeness

When we examine bidirectionality in connection with the dynamic semantics, several questions arise. The first is: How should preservation and progress be formulated? Perhaps we could formulate preservation as

(1) If $\vdash e \uparrow \tau$ and $e \mapsto e'$ then $\vdash e' \uparrow \tau$
(2) If $\vdash e \downarrow \tau$ and $e \mapsto e'$ then $\vdash e' \downarrow \tau$

But this becomes very messy; even proving that $(\lambda x.e)v \mapsto \{v/x\}e$ preserves types is nasty. The type τ of $(\lambda x.e)v$ is inferred, so we have case (1), but the premise of (Lam) is a checking judgment, so we don't have $x{:}\sigma \vdash e \uparrow \tau$ which would lead (by substitution) to $\{v/x\}e \uparrow \tau$.

Besides, e may have type annotations. While we can certainly write a rule

$$\frac{}{(e : \tau) \mapsto e}$$

it doesn't remotely correspond to any reasonable model of computation.

The formulation

If $\vdash e \uparrow \tau$ or $\vdash e \downarrow \tau$ and $e \mapsto e'$ then $\vdash e' \uparrow \tau$ or $\vdash e' \downarrow \tau$

might be correct, but it suggests that we don't actually care about the direction. Which is indeed the case: we use bidirectionality so we can write the typechecker; it has nothing to do with running the program. Indeed, since type annotations are (in some cases) essential to bidirectional typing, and types should not matter at runtime, there seems to be a gulf between bidirectional typing and dynamic semantics.

Since we know how to state (and prove) preservation and progress for a non-bidirectional type system, why not have a non-bidirectional system as well, and show that we can get from a bidirectional typing to a typing in that system? The non-bidirectional system I have in mind is simply the bidirectional system without (Anno) and with all $\uparrow$, $\downarrow$ changed to $:$. Formally:

Theorem 1 (Soundness). *If $\vdash e \downarrow \tau$ or $\vdash e \uparrow \tau$ then $\vdash |e| : \tau$ where $|e|$ is e with all type annotations erased.*

Proving this is straightforward (after generalizing to an arbitrary context Γ). I call it *soundness* because it says that the bidirectional system is

sound with respect to the non-bidirectional system: anything derivable in the first is derivable in the second (after erasing type annotations).

Do we have *completeness*? No. As a counterexample,

$$\texttt{case}(\texttt{inl}(5), x_1.0, x_2.1)$$

is well-typed in the non-bidirectional system, but not in the bidirectional system.

4.4 Polymorphism

$$\frac{\Gamma, t\ \mathsf{type} \vdash e \downarrow \sigma}{\Gamma \vdash \mathtt{Fun}(t.e) \downarrow \forall t.\sigma}\ \text{(Typefun)} \qquad \frac{\Gamma \vdash e \uparrow \forall t.\sigma \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathtt{Inst}(e, \tau) \uparrow \{\tau/t\}\sigma}\ \text{(Inst)}$$

Exercise: derive

$$\vdash \mathtt{Fun}(t.\lambda x.\, x) \downarrow \forall t.t \to t$$

4.5 Other Type Constructors

See the Assignment 6 handout.

Supplementary Notes on Records

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 18
October 29, 2002

A common generalization of the notion of a product is a *record*. A record is like a tuple, except that the components are named explicitly by a *record label*. All labels in a record must be distinct. Records can also be used as the foundation for object-oriented programming idioms in a functional language. In the case, an object would be represented as a record, and a record label would be either a field name or a method name.

We begin by studying records in themselves; later we consider how to model some features of objects. We extend the type system by *record types* that we denote by ρ; we use l to denote record labels.[1]

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \ldots \mid \{\rho\} \\ \text{Record Types} & \rho & ::= & \cdot \mid l{:}\tau, \rho \end{array}$$

We extend expressions to allow the formation of records, denoted by r, and also the selection of a field from a record, written as $e.l$ for a record label l.

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \ldots \mid \{r\} \mid e.l \\ \text{Records} & r & ::= & \cdot \mid l{=}e, r \end{array}$$

We sometimes use parentheses to enclose record types or record so the scope of the ',' is more clearly visible. Such parentheses are not properly part of the syntax of the language. We have a new typing judgment $r : \rho$, used in the following rules.

[1] Not to be confused with memory locations that we use to study mutable references.

$$\frac{\Gamma \vdash r : \rho}{\Gamma \vdash \{r\} : \{\rho\}} \qquad \frac{\Gamma \vdash e : \{\rho\} \quad \rho = \rho_1, l{:}\tau, \rho_2}{\Gamma \vdash e.l : \tau}$$

$$\frac{}{\Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash r : \rho}{\Gamma \vdash (l{=}e, r) : (l{:}\tau, \rho)}$$

Note that the field selection operation $e.l$ will always yield a unique answer on well-typed records. This is because labels in a record must be unique. The order of the fields in a record is significant, although we discuss below how this can be relaxed using *exchange subtyping* for records.

In this notation, the empty record expression corresponds to the unit type. Note that there is a minor ambiguity in that the empty record and its type are both denoted by '$\cdot$', that is, $\vdash \{\cdot\} : \{\cdot\}$. As usual, we omit a leading '$\cdot$' in a record.

A pair $\texttt{pair}(e_1, e_2)$ can represented by the record $\{1{=}e_1, 2{=}e_2\}$. Then the first and second projection are defined by $\texttt{fst}(e) = e.1$ and $\texttt{snd}(e) = e.2$, respectively. This is the approach taking in Standard ML, using the notation $\#l(e)$ instead of $e.l$.

Records in this form may make code more readable, because instead of writing $\texttt{fst}(e)$, we write $e.l$, where l is presumably a meaningful label. However, a much greater advantage can be derived from records if we add rules of subtyping. Before we describe this, we give the operational semantics for records. There are two new judgments, $r \ \ \mathsf{value}$ and $r \mapsto r'$. A record is a value if all fields are values, and we evaluate the components of a record from left to right.

$$\frac{r \mapsto r'}{\{r\} \mapsto \{r'\}}$$

$$\frac{}{(\cdot) \ \ \mathsf{value}} \qquad \frac{v \ \ \mathsf{value} \quad r \ \ \mathsf{value}}{(l{=}v, r) \ \ \mathsf{value}}$$

$$\frac{e \mapsto e'}{(l{=}e, r) \mapsto (l{=}e', r)} \qquad \frac{r \mapsto r'}{(l{=}v, r) \mapsto (l{=}v, r')}$$

$$\frac{e \mapsto e'}{e.l \mapsto e'.l} \qquad \frac{r \ \ \mathsf{value} \quad r = (r_1, l{=}v, r_2)}{\{r\}.l \mapsto v}$$

The progress and type preservation theorems now also have to account for the new judgment, but this is entirely straightforward and omitted here.

There are three forms of subtyping that can be considered together or in isolation: *depth subtyping*, *width subtyping* and *exchange subtyping*.

The general rule passes from ordinary types to record types and is common to all forms of subtyping.

$$\frac{\rho \leq \rho'}{\{\rho\} \leq \{\rho'\}} \leq \text{record}$$

Depth subtyping. This is the idea we used for product subtyping, applied to records. Subtyping is co-variant in all fields of a record.

$$\frac{\tau \leq \tau' \quad \rho \leq \rho'}{(l{:}\tau, \rho) \leq (l{:}\tau', \rho')} \leq_d \textit{field} \qquad \frac{}{(\cdot) \leq (\cdot)} \leq_d \textit{empty}$$

We do not show formally how to construct coercions, but consider the following sample coercion.

$$\lambda r.\{x{=}\mathsf{itof}(r.x), y{=}\mathsf{itof}(r.y)\} : \{x{:}\mathsf{int}, y{:}\mathsf{int}\} \leq \{x{:}\mathsf{float}, y{:}\mathsf{float}\}$$

Width subtyping. The idea of width subtyping is that we can always coerce from a record with more fields to a record with fewer by dropping some extra fields. We separate out the idea of exchange and allow fields to be dropped only at the end of a record.

$$\frac{\rho \leq \rho'}{(l{:}\tau, \rho) \leq (l{:}\tau, \rho')} \leq_w \textit{field} \qquad \frac{}{\rho \leq (\cdot)} \leq_w \textit{empty}$$

Note that the $\leq_w$*field* rule does not allow any subtyping on the field itself—this would require the combination of width and depth subtyping.

We can give width subtyping both a subset and a coercion interpretation. The subset interpretation would say that a value of type $\{\rho\}$ can be any extension of $\{\rho, \rho'\}$: the additional fields ρ' are simply ignored. This is common in object-oriented languages. It requires a so-called boxed representation where every object is simply a pointer to the actual object, so that for the purpose of argument passing, every record has the same size.

The coercion interpretation would explicitly shorten the object, which is not usually practical. Nonetheless, under this interpretation we might have a coercion such as

$$\lambda r.\{x{=}r.x, y{=}r.y\} : \{x{:}\mathsf{float}, y{:}\mathsf{float}, c{:}\mathsf{int}\} \leq \{x{:}\mathsf{float}, y{:}\mathsf{float}\}$$

Exchange subtyping. This means we can reorder the fields. We formalize this by allowing corresponding fields to be picked out from anywhere in the middle of the record.

$$\frac{(\rho_1, \rho_2) \leq (\rho_1', \rho_2')}{(\rho_1, l{:}\tau, \rho_2) \leq (\rho_1', l{:}\tau, \rho_2')} \leq_x \textit{field} \qquad \frac{}{(\cdot) \leq (\cdot)} \leq_x \textit{empty}$$

An implementation that allows exchange subtyping would typically sort the fields alphabetically by field name.

It is straightforward to construct type systems for record that combine depth, width, and exchange subtyping. We will see what is needed in order to model some object-oriented features, following Chapter 18 of Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002. We only sketch the rationale and implementation below; for more detail see the above reference.

Objects. As a very first approximation we think of an object as a record with some internal state. This internal state is *encapsulated* in that it can only be accessed through the visible fields of the method. We use a simple counter as an example.

```
Counter = {get:1 -> int, inc:1 -> 1};
c : Counter =
    let x = ref 1
     in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

In the terminology of object-oriented languages, `x` is a private field, accessible only to the methods `get` and `inc`.

We can increment and then read the counter by sending messages to `c`. This is accomplished by calling the functions in the fields of `c`.

```
(c.inc(); c.inc(); c.get()); ↦* 3
```

Object Generators. We can package up the capability of creating a new counter object.

```
newCounter : 1 -> Counter =
  λ_:1.
    let x = ref 1 in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

Subtyping. We can easily create an object with more methods. Width subtyping allows us to use the object with more methods in any place the the object with fewer methods is required.

```
ResetCounter =
  {get:1 -> int, inc:1 -> 1, reset:1 -> 1};
newResetCounter : 1 -> ResetCounter =
  λ_:1
    let x = ref 1 in
      {get = λ_:1. !x,
       inc = λ_:1. x := !x+1,
       reset = λ_:1. x := 1}
    end;
```

Grouping Instance Variables. The instance variable x was just a single variables; it is more consistent with the approach to group them into a record. The modification is completely straightforward.

```
Counter = {get:1 -> int, inc:1 -> 1};
newCounter : 1 -> Counter =
  λ_:1.
    let r = {x = ref 1} in
        {get = λ_:1. !(r.x),
         inc = λ_:1. r.x := !(r.x)+1}
    end;
```

Simple Classes. We can extract the instance variables and make them a parameter of the instance creation mechanism. This will allow us to give a simple model of subclassing and inheritence.

```
CounterRep = {x : int ref};
Counter = {get:1 -> int, inc:1 -> 1};
CounterClass : CounterRep -> Counter =
  λr:CounterRep.
   {get = λ_:1. !(r.x),
    inc = λ_:1. r.x := !(r.x)+1};
newCounter : 1 -> Counter =
  λ_:1. let r = x=ref 1 in counterClass r end;
```

The possibility of a shared representation allows us to create an instance of the `ResetCounter` subclass by first constructing a `Counter`.

```
ResetCounterClass : CounterRep -> ResetCounter =
  λr:CounterRep.
    let super = counterClass r in
     {get = super.get,
      inc = super.inc,
      reset = λ_:1. r.x := 1}
    end;
newResetCounter : 1 -> ResetCounter
  λ_:1. let r = {x=ref 1} in resetCounterClass r end;
```

Adding Instance Variables. So far, subtyping only allows us to use instances of a subclass where instances of a superclass are required. When we add instance variables, we need it in another place, namely where the representation of the instance of the superclass is created.

```
BackupCounter =
  {get:1 -> int, int:1 -> 1, reset:1 -> 1,
   backup:1 -> 1};
BackupCounterRef =
  {x:int ref, b:int ref};
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in    % subtyping here
      {get = super.get,
       inc = super.inc,
       reset = λ_:1. r.x := !(r.b),
       backup = λ_:1. r.b := !(r.x)}
    end;
```

As we can see, references to instances of the superclass are easy. But references to the methods of the class itself within the method are somewhat tricker, but an essential technique in object-oriented languages. We discuss this in the next lecture.

Supplementary Notes on Objects

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 19
October 31, 2002

In this lecture we extend the encoding of objects using records and subtyping from the previous lecture to allow open recursion through self. We still follow Chapter 18 of Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002.

Classes with Self. First, we show how "self", that is, invoking of methods part of the current objects, can be encoded directly. This technique does *not* model open recursion (also called late binding of self).

The basic idea of this first approach is to use a fixed point construct in order to allow the methods of in an object to refer to the object itself. The fixpoint operator is orthogonal to all other operators in the language and can be defined with

$$\frac{\Gamma, f{:}\tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f.e : \tau} \qquad \frac{}{\mathsf{fix}\, f.e \mapsto \{\mathsf{fix}\, f.e/x\}e}$$

There are no new values. Type preservation requires that all three types involved in the definition of fix be the same.

Now we take a slight modification of the previous example, extending the class to allow `get`, `set`, and `inc` methods. Internally, the `inc` method refers to `set` and `get` instead of accessing the private fields directly.

```
CounterRep = {x:int ref};
SetCounter = {get:1 -> int, set:int -> 1, inc:1 -> 1};
setCounterClass : CounterRep -> SetCounter =
  λr:CounterRep.
    fix self:SetCounter.
      {get = λ_:1. !(r.x),
       set = λi:int. r.x := i,
       inc = λ_:1. self.set (self.get() + 1)};
newSetCounter : 1 -> SetCounter =
  λ_:1. let r = {x = ref 1} in setCounterClass r end;
```

To see this representation in action, we apply the operational semantics to `newSetCounter().inc()` in the empty store. Each line constitutes a state of the machine, although we skip several intermediate steps.

```
<., newSetCounter().inc()>
<., let r = {x=ref 1} in setCounterClass r end.inc()>
<c=1, (setCounterClass {x=c}).inc()>
<c=1, (fix self.
        {..., inc=λ_:1. self.set(self.get()+1)}).inc()>
```

At this point we abbreviate

```
s = fix self. {..., inc = λ_:1. self.set(self.get()+1)}
```

and continue execution with

```
<c=1, {...,inc=λ_:1. s.set(s.get()+1)}.inc()>
<c=1, λ_:1. s.set(s.get()+1)()>
<c=1, s.set(s.get()+1)>
<c=1, s.set((λ_:1. !({x=c}.x))()+1)>
<c=1, s.set(!({x=c}.x)+1)>
<c=1, s.set(2)>
<c=2, ()>
```

Open Recursion through Self. The previous encoding is perfectly adequate, yet it does not model a feature available in many object-oriented languages, namely open recursion. This feature means that in a subclass of `SetCounter` that overrides `set` but not `inc`, the references to `self` will be to the `set` and `get` methods of the *subclass*. This feature is somewhat unfortunate, because it breaks encapsulation: as we will see after we have

modeled the feature, client code will depend on internals of the implementation of a superclass.

To model this feature we move the recursion outside the object itself to the place where it is created.

```
setCounterClass : CounterRep -> SetCounter -> SetCounter =
  λr:CounterRep.
    λself:SetCounter.
      {get = λ_:1. !(r.x),
       set = λi:int. r.x := i,
       inc = λ_:1. self.set (self.get() + 1)};
newSetCounter : 1 -> SetCounter =
  λ_:1. let r = {x = ref 1}
           in fix self. setCounterClass r self end;
InstrCounterRep = {x:int ref, a:int ref}
InstrCounter =
  {get:1 -> int, set:int -> 1,
   inc:1 -> 1, accesses:1 -> int}
instrCounterClass :
  InstrCounterRep -> InstrCounter -> InstrCounter =
  λr:InstrCounterRep.
    λself:InstrCounter.
      let super = setCounterClass r self
       in
          {get = super.get,
           set = λi:int. (r.a := !(r.a)+1; super.set i),
           inc = super.inc,
           accesses = λ_:1. !(r.a)}
      end;
```

The previous problem has now been solved, yet a new problem has arisen, because creating objects of type `instrCounterClass` By using the standard technique of unit-abstractions, we can overcome this problem (see Chapter 18.11 of Pierce's book). We will not go into those details here.

We close the lecture by exhibiting that this form of late binding of self breaks encapsulation and is extremely dangerous when writing a library. It means that the behavior of a subclass can depend on specifics of the (supposedly invisible!) internal of the library class. The example we show here is taken from Item 14 of Joshua Bloch: *Effective Java*, Addison-Wesley, 2001,

The following code uses inheritance inappropriately, as the example at the end shows.

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet () {
    }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

Now the following sequence

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[]
          {"Snap", "Crackle", "Pop"}));
s.getAddCount();
```

may return either 3 or 6, depending on whether the library implementation of `addAll` has internal calls to `add` or not.

To the writer of this library this means he must either fully document the internal call patterns of the library, or risk breaking a lot of client code when improving internal data structures. Bloch suggests that it is often better to *prohibit* inheritance, for example, making constructors private, and using *composition* instead of inheritance. Precisely the same technique would be used in Standard ML's module system to obtain the effect by inheritance. First, the corresponding Java code.

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private in addCount = 0;
    public InstrumentedSet(Set s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
    // Forwarding methods
    public void clear()               { s.clear ();           }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    // ...
    public String toString ()         { return s.toString();  }
}
```

In the place of `// ...` all the relevant public methods of `s` are exported again. In ML we would use a wrapper functor instead (assuming we really wanted the implementation of `Set` to be ephemeral):

```
signature InstrumentedSet =
sig
  include Set
  val getAddCount : unit -> int
end;
functor InstrumentWrapper (structure S : Set)
      : InstrumentedSet =
struct
  val addcount = ref 0
  open S
  fun add(o) = (addcount := !addcount+1; S.add(o))
  fun addAll(c) = (addcount := !addcount+List.length(c); S.addAll(c))
  fun getAddCount() = !addCount
end;
structure InstrumentedSet = InstWrapper (structure S = Set);
```

We would like to emphasize that open recursion is indeed an anti-modularity feature that breaks encapsulation. Any programmer that prepares libraries in a language like Java should be aware of this, and know how to avoid its pitfalls that are sometime difficult to detect.

There are many other features of object-oriented languages that we have not yet modeled. We return to some of them in the next lecture. Here we would like to discuss *overloading*. In Java it refers to the fact that a method name can be reused, as long as all its argument types are different. We only discuss it on simpler examples, namely the overloading of addition. Assume we have two internal functions, $+_{int}$ and $+_{float}$ that add integers and floating point numbers.

In the concrete syntax of the language, we would like to use $+$ and let the type checker sort out which of the two versions of addition should be used. Intersection types, in conjunction with subtyping, allow precisely that. We write $\tau \wedge \sigma$ for the intersection between τ and σ. We have the following rules:

$$\frac{\Gamma \vdash v : \tau_1 \quad \Gamma \vdash v : \tau_2 \quad v \ \mathsf{value}}{\Gamma \vdash v : \tau_1 \wedge \tau_2} \wedge I$$

$$\frac{\Gamma \vdash e : \tau_1 \wedge \tau_2}{\Gamma \vdash e : \tau_1} \wedge E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \wedge \tau_2}{\Gamma \vdash e : \tau_2} \wedge E_2$$

The restriction of the intersection introduction rule to values is necessary for soundness in the presence of mutable references. The counterexample (which we do not show here) echoes the related counterexample that requires the value restriction in Standard ML.

Together with the introduction and elimination rules, we also have three subtyping rules, working on the left or the right hand side.

$$\frac{\tau \leq \sigma_1 \quad \tau \leq \sigma_2}{\tau \leq \sigma_1 \wedge \sigma_2} \wedge R$$

$$\frac{\tau_1 \leq \sigma}{\tau_1 \wedge \tau_2 \leq \sigma} \wedge L_1 \qquad \frac{\tau_2 \leq \sigma}{\tau_1 \wedge \tau_2 \leq \sigma} \wedge L_1$$

With these concepts, we can now declare

$$+ : (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \wedge (\mathsf{float} \to \mathsf{float} \to \mathsf{float})$$

Below are several judgments that check with these declarations.

$$\begin{array}{rcl} 3 + 4 & : & \mathsf{int} \\ 3.0 + 4 & : & \mathsf{float} \\ 3 + 4.0 & : & \mathsf{float} \\ 3.0 + 4.0 & : & \mathsf{float} \end{array}$$

We use the $\wedge E_1$ rule in the first case, and $\wedge E_2$ to extract the appropriate type for $+$ in each judgment. As before, we also need to coerce the integer arguments to floating point numbers, in the middle two examples.

Since we are using a coercion interpretation of subtyping here, we have to show how to interpret intersection types. From the primitive function $+$ we can see that a constant of intersection type actually corresponds to a pair of two functions

$$\mathtt{pair}(+_{int}, +_{float}) \quad : \quad (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \times (\mathsf{float} \to \mathsf{float} \to \mathsf{float})$$

From this we can extend interpretation through the whole type hierarchy. We achieve this by annotating a type derivation by the fully explicit expression it generates. We also extend the annotation of the subtyping rules to account for the new coercion. The judgment is $\Gamma \vdash e : \tau \implies e'$ where e' is an explicit term without any uses of subtyping or intersection types. It has type τ' which is generated from τ by replacing intersections ($\wedge$) by products. We write $\overline{\tau}$ for this operation.

$$\frac{\Gamma \vdash v : \tau_1 \implies v_1 \quad \Gamma \vdash v : \tau_2 \implies v_2 \quad v \ \mathsf{value}}{\Gamma \vdash v : \tau_1 \wedge \tau_2 \implies \mathtt{pair}(v_1, v_2)} \wedge I$$

$$\frac{\Gamma \vdash e : \tau_1 \wedge \tau_2 \implies e'}{\Gamma \vdash e : \tau_1 \implies \mathtt{fst}(e')} \wedge E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \wedge \tau_2 \implies e'}{\Gamma \vdash e : \tau_2 \implies \mathsf{snd}(e')} \wedge E_2$$

For most other constructs the propagation of the explicitly typed term is straightforward. We show only a few further cases

$$\frac{(\Gamma = \Gamma_1, x{:}\tau, \Gamma_2)}{\Gamma \vdash x : \tau \implies x}\ var \qquad \frac{\Gamma \vdash e : \tau \implies e' \quad f : \tau \leq \sigma}{\Gamma \vdash e : \sigma \implies f(e')}\ sub$$

Subtyping introduces no new ideas.

$$\frac{f_1 : \tau \leq \sigma_1 \quad f_2 : \tau \leq \sigma_2}{\lambda x.\mathtt{pair}(f_1(x), f_2(x)) : \tau \leq \sigma_1 \wedge \sigma_2}\ \wedge R$$

$$\frac{f_1 : \tau_1 \leq \sigma}{\lambda x.f_1(\mathtt{fst}(x)) : \tau_1 \wedge \tau_2 \leq \sigma}\ \wedge L_1 \qquad \frac{f_2 : \tau_2 \leq \sigma}{\lambda x.f_2(\mathtt{snd}(x)) : \tau_1 \wedge \tau_2 \leq \sigma}\ \wedge L_2$$

Recall that $\overline{\tau}$ replaces intersections by products. We extend this operation to context by applying it to each type declaration.

Theorem 1 (Coercions)

(i) If $\tau \leq \sigma$ then $f : \tau \leq \sigma$ for some f.

(ii) If $\Gamma \vdash e : \tau$ then $\Gamma \vdash e : \tau \implies e'$ for some e' with $\overline{\Gamma} \vdash e' : \overline{\tau}$.

Proof: By rule induction on the given derivations. ■

Note that we also want f and e' to be unique. However, this can only be true extensionally, since subtyping derivations (and therefore typing derivations) are not uniquely determined. In other words, we do not formalize here the all-important property of *coherence*.

We cannot execute or define the operational semantics directly on a source expression e, because the process of inserting the coercions performs the overloading resolution. As in Java, this process is completely static, that is, happens before the program is ever executed. The above should therefore be considered a reasonable model for the kind of overloading present in object-oriented languages. Other languages, such as Haskell, permit overloading, but overloading is not resolved until run-time—this requires different techniques for both proving progress and preservation than we discuss here.

Supplementary Notes on Dynamic Typing

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 20
November 5, 2002

So far, we have been working with type systems were all checking was static, and types were not needed at run-time. In this lecture we investigate the consequences of loosening this assumption. A language in which types are used at run-time is called *dynamically typed*. Examples of dynamically typed languages are Lisp, Scheme, and Java. There is an excellent treatment of dynamic types in [Ch. 24]. We emphasize a few complementary points here.

We begin with the extreme point of no static type-checking at all. We simply take a program written in MinML and run it without type-checking it first. Clearly, this would violate progress since a term such as $\mathtt{apply}(1, 1)$ is neither a value, nor can it make a step. In order to obtain a sensible language, we need to extend our computational rules to check for such stuck states and raise a run-time type error. We show here only the rules for function application. First, the usual rules, written (implicitly) under the assumption of type-correctness.

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

$$\frac{(v_1 = \mathtt{fun}(f.x.e_1)) \quad v_2 \ \mathsf{value}}{\mathtt{apply}(v_1, v_2) \mapsto \{v_2/x\}\{v_1/f\}e_1}$$

Next, the rules to handle the case of a run-time type error. The first one signals the actual error, the others propagate the error outward.

$$\frac{v_1 \text{ value} \quad (v_1 \neq \texttt{fun}(f.x.e_1)) \quad v_2 \text{ value}}{\texttt{apply}(v_1, v_2) \mapsto \texttt{error}}$$

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

The rest of the operational semantics should be extended in a similar way. In particular, there need to be additional rules for primitive operations and other elimination forms (conditionals, projections, case expressions, etc.) in the case of a dynamic type error.

In the resulting language, we can write and execute expressions such as

```
if true then 1 else λx. x  ↦  1
[1, true, λx. x]  value
(hd (tl (tl [1, true, λx. x]))) 3 ↦* 3
```

where we used ML-style notations for lists.

With these additions, we can recover preservation and progress. We assume here that we still want to statically compile the program, so free variables are still not permitted (e must be closed, that is, $FV(e) = \{\}$). Preservation is somewhat trivialized.

Theorem 1 (Preservation)
If e is closed and $e \mapsto e'$ then e' is closed.

It is also posssible to talk about expressions e that happen to be well-typed. In that case, evaluation preserves types, since we have only added rules to the operational semantics in a conservative way.

Theorem 2 (Progress)
If e is closed then either

(i) e value*, or*

(ii) $e = \texttt{error}$*, or*

(iii) $e \mapsto e'$ *for some* e'*.*

The main conceptual cost of dynamic typing is the inability to discover errors early: code that does not happen to be executed can have lurking bugs that would be obvious to a type-checker. However, there is also an

implementation cost. We cannot simply compile a program with the same strategy as may be possible for a statically typed language, because we must be able to perform the run-time type checks. Fortunately, this is not as bad as it sounds, since we do not need to have the precise type of a function, we only need to know where it is in fact a function or not.

In order to make tags explicit, we can change the syntax of our language; see [Ch. 24] for the details. Here we show that we can make such tags explicit in a statically typed language. The small price we pay for this is that now the user program or library will have to do some tag-checking. But this means we can tag-check precisely where necessary, instead of everywhere in the program.

In ML, we would write such a tagged datatype as

```
datatype tagged =
    Int of int
  | Bool of bool
  | Fun of tagged -> tagged -> tagged
```

Note that we do not explicitly represent bound variables, as they are mapped to bound variables in ML. This is an application of the idea of higher-order abstract syntax.

As a reminder, here is how we would write this type using plain recursive types instead.

$$\mathsf{tagged} = \mu t.\mathsf{int} + \mathsf{bool} + (t \to t \to t)$$

Instead of named tags, as in the datatype declaration above, we use compositions of inl and inr as tags.

Now the elimination forms become functions on tagged representations. We show one primitive operator, conditional, and application.

```
exception TypeError
fun checkedMult (Int(n), Int(m)) = Int(n*m)
  | checkedMult _ = raise TypeError
fun checkedIf (Bool(true), e1, e2) = e1 ()
  | checkedIf (Bool(false), e1, e2) = e2 ()
  | checkedIf _ = raise TypeError
fun checkedApply (v1 as Fun(g), v2) = g v1 v2
  | checkedApply _ = raise TypeError
```

Heterogeneous lists now become lists of tagged data. This is in fact exactly the same as the representation in a purely dynamically typed language, except that the tagging is visible to the programmer. For example, the following are all well-typed and execute as expected:

```
val hetList : tagged list =
  [Int 1, Bool true, Fun (fn _ => fn x => x)];
val f : tagged = hd(tl(tl(hetList)));
val x : tagged = checkedApply (f, Int(3));
```

The non-terminating self-application example can also be written quite easily using checked application. Note that even though functions can in principal be recursive in our encoding, we do not use this feature here to give a correct implementation of the dynamically typed $(\lambda x.x\,x)\,(\lambda x.x\,x)$.

```
val omega : tagged =
  Fun (fn _ => fn x => checkedApply (x, x));
checkedApply (omega, omega);
```

As expected, this last expression diverges.

It is also interesting to consider if we can perhaps use subtyping to allow us to write heterogenous lists. For example, if we had a universal type $\top$ that includes all values, one might expect

```
[1, true, λx. x] : ⊤ list
```

In order to see if this is indeed the case, we consider the laws that $\top$ should satisfy. First, every value should have type $\top$. Second every type should be a subtype of $\top$.

$$\frac{v \text{ value}}{\Gamma \vdash v : \top} \qquad \frac{}{\tau \leq \top}$$

There are no other rules regarding $\top$. Clearly, it is necessary to require v to be a value in the first rule in order to save the progress theorem.

Now, indeed, we have

```
[1, true, λx. x] : ⊤ list
```

However, we find we cannot *use* such a list in a non-trivial way. For example

```
val hetList : ⊤ list = [1, true, λx. x];
val f : ⊤ = hd(tl(tl(hetList)));
```

Now the application `f 3` would not be well-typed, because `f` is not known to be a function, only a value of type $\top$.

In order to use `f`, we must introduce a *downcast* operator into the language, that allows us to check explicitly *at run-time* if a given value has a specified type, and raise an error otherwise. We could then write

```
((int -> int)f) 3
```

and it would type-check. Of course, at this point we realize we haven't made any progress, because the function still must be tagged with its type in order to verify the correctness of the downcast at run-time. In fact, we would need to keep more information to verify the precise form of the function's type, rather than just the information that it is a function.

Note that his form of downcast is very different from an unsafe version of cast where $(\tau)e$ will be treated as if it has type τ, regardless of the actual type of the value of v. In a language like C this cannot be repaired, because data are not tagged and no run-time checking of tags is possible.

What would be the coercion interpretation of the $\top$ type? Recall that intersection types are interpreted as pairs. If we think of $\top$ as a 0-ary intersection, it should logically be interpretated as a 0-ary product, namely the unit type.

$$\overline{\lambda x.\langle\,\rangle : \tau \leq \top}$$

Intuitively, this is also meaningful: the coercion from any type to $\top$ is unique (the constant function) and coherence is preserved. Knowing that $v : \top$ carries no information.

This points out an important property of the coercion interpretation of subtyping: since we run the program *after* all coercions have been applied, any term that was assigned type $\top$ via a subtyping coercion may not be executed at all! This is nonetheless consistent since essentially only values have type $\top$ directly. But this means that we cannot implement downcasting in a coercion interpretation of subtyping. Again, intuitively this make sense: since a coercion from τ to σ where $\tau \leq \sigma$ loses information,

we cannot in general recover an element of the original type τ if we try to downcast a value of type σ.

On the other hand, under the subset interpretation of subtyping, a coercion is useless, since all coercions will be the identity: if a value of type τ *is* a value of type σ then there is no need to apply a coercion. In that case downcasting as a run-time operation that may fail makes sense: in a downcast $(\tau)e$ for $e : \sigma$ we evaluate e, and then verify if it has type τ. The latter operation will, of course, require tags or some other method to check that a given value has a specified type at run-time.

These observations help to explain why objected-oriented languages such as Java, which rely heavily on downcasting, have a subset interpretation of subtypes that arise from subclassing. In such languages objects are tagged with their class, which makes it quite efficient to implement downcasts or the related `instanceOf` operation. Of course, it is critical that viewing an object of a class as an element of the superclass does *not* apply a coercion, dropping extra fields, because later downcasts could not undo this damage.

Also, in Java there is a class `Object` which is a superclass of any other class. This almost corresponds to $\top$, except that Java also has primitive types such as `int` or `float`. Given that Java performs coercions on `int` and `float` we could summarize the situation as coercive subtyping on primitive types and subset subtyping on objects.

Supplementary Notes on Futures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 21
November 7, 2002

In this lecture we first examine a technique to specify the operational semantics for 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* 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 lazy evaluation 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. We separate here recursion from functions.

$$\frac{\langle H, e_1\rangle \mapsto \langle H', e_1'\rangle}{\langle H, \mathtt{apply}(e_1, e_2)\rangle \mapsto \langle H', \mathtt{apply}(e_1', e_2)\rangle}$$

$$\frac{}{\langle H, \mathtt{apply}(\lambda x.e_1, e_2)\rangle \mapsto \langle (H, l{=}e_2), \{l/x\}e_1\rangle}$$

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 l, which would lead to an infinite loop. This particular form of infinite loop is called a *black hole* can be statically detected, while other forms of nontermination 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 lazy evaluation 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.

$$\overline{\langle H, \mathsf{fix}\, x.e\rangle \mapsto \langle (H, l{=}\{l/x\}e), l\rangle}$$

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

$$\frac{\langle (H_1, l{=}\bullet, H_2), e\rangle \mapsto \langle (H_1', l{=}\bullet, H_2'), e'\rangle}{\langle (H_1, l{=}e, H_2), l\rangle \mapsto \langle (H_1', l{=}e', H_2'), l\rangle}$$

$$\overline{\langle (H_1, l{=}\bullet, H_2), l\rangle \mapsto \langle (H_1, l{=}\bullet, H_2), \mathsf{BlackHole}\rangle}$$

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.

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 lazy evaluation as shown above. The first is that a future is a bona fide value. This is important because unlike lazy evaluation, 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.

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 lazy 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 $l{=}v$.

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.

$$\frac{\langle (H_1, H_2), e\rangle \mapsto \langle H', e'\rangle}{(H_1, l{=}e, H_2) \mapsto (H', l{=}e')} \qquad \frac{}{l \ \ \mathsf{value}}$$

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 `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. This possibility is pursued in Assignment 8.

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
```

Stating a preservation theorem for MinML with futures is not difficult. However, proving progress is tricky, because typing alone among multiple processes does not rule out the possibility of a deadlock. Instead, we must assume a partial order among processes so that minimal elements in the order cannot block. It is important that his order is maintained during

computation and that it remains an order, that is, remains acyclic. Again, we will not pursue this direction further.

Supplementary Notes on Program Equivalence

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 22
November 12, 2002

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 argument, 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 out 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

$$\begin{array}{lcl} id_1 & = & \lambda x.x \\ id_3 & = & \mathsf{fun}\ f(x)\ \mathsf{is\ if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1) + 1 \end{array}$$

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 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.

Some care must be taken in general to define observational equivalence correctly with respect to what is observable. For example, assume we have a call-by-name language. In that case, we have to change the definition of observational equivalence as follows.

$$\begin{array}{lll} e \cong e' : \tau & \text{iff} & \text{either } e \Uparrow \text{ and } e' \Uparrow \\ & & \text{or } e \mapsto^* v \text{ and } e' \mapsto^* v' \text{ with } v \simeq v' : \tau \end{array}$$

$$\begin{array}{lll} v \simeq v' : \mathsf{int} & \text{iff} & v = v' = n \text{ for an integer } n. \\ v \simeq v' : \mathsf{bool} & \text{iff} & v = v' = \mathsf{true} \text{ or } v = v' = \mathsf{false} \\ v \simeq v' : \tau_1 \to \tau_2 & \text{iff} & \text{for all } e_1 \cong e_1' : \tau_1 \text{ we have } v\, e_1 \cong v'\, e_1' : \tau_2 \end{array}$$

The only change to the definition in the call-by-value case in the last clause. Note that now it is necessary to verify the equivalence of $v\, e_1$ and $v\, e_1'$ for arbitrary equivalent *expressions* e_1 and e_1'. Checking this just for values is no longer enough. For example in a call-by-value language we have

$$\lambda x.0 \simeq \lambda x.x - x : \mathsf{int} \to \mathsf{int}$$

because both functions return 0 when applied to arbitrary integers. However, in a call-by-name language the two differ

$$\lambda x.0 \not\simeq \lambda x.x - x : \mathsf{int} \to \mathsf{int}$$

since $(\lambda x.0)\,(\mathsf{fix}\, y.y) \mapsto 0$ while $(\lambda x.x - x)\,(\mathsf{fix}\, y.y)$ diverges.

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 $P \stackrel{\alpha}{\longrightarrow} P'$ meaning that P transitions in one step to P' exhibiting action α.

$$\frac{}{M + \alpha.P + N \stackrel{\alpha}{\longrightarrow} P}\ \text{Sum} \qquad \frac{(A \stackrel{\text{def}}{=} P_A) \quad P_A \stackrel{\alpha}{\longrightarrow} P'}{A \stackrel{\alpha}{\longrightarrow} P'}\ \text{Def}$$

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 described as a sequential process as follows:

$$A \quad \stackrel{\text{def}}{=} \quad 2\text{p}.(\overline{\text{tea}}.A + 2\text{p}.\overline{\text{coffee}}.A)$$

[1]This example is taken from Robin Milner's book on *Communicating and Mobile Processes: the π-Calculus*, Cambridge University Press, 1999.

The vending machine has three states: an initial state A (in which it only waits for the input of $2\mathrm{p}$), a state B where we can either get the $\overline{\mathrm{tea}}$, or put in another $2\mathrm{p}$, and a state C where can only ge the $\overline{\mathrm{coffee}}$. We can make this explicit with this alternative definition

$$\begin{array}{lcl} A & \stackrel{\mathrm{def}}{=} & 2\mathrm{p}.B \\ B & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{tea}}.A + 2\mathrm{p}.C \\ C & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{coffee}}.A \end{array}$$

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.

Supplementary Notes on Concurrent Processes

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 23
November 14, 2002

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 $\overline{\mathrm{b}}$, returning to state A. Process B performs an input action b followed by an output action $\overline{\mathrm{c}}$, returning to state B upon completion.

$$\begin{array}{lcl} A & \stackrel{\text{def}}{=} & \mathrm{a}.\overline{\mathrm{b}}.A \\ B & \stackrel{\text{def}}{=} & \mathrm{b}.\overline{\mathrm{c}}.B \end{array}$$

We assume we start with A and B operating concurrently, that is, in state

$$A \mid B$$

Now we can have the following sequence of transitions:

$$A \mid B \xrightarrow{\mathrm{a}} \overline{\mathrm{b}}.A \mid \mathrm{b}.\overline{\mathrm{c}}.B \longrightarrow A \mid \overline{\mathrm{c}}.B \xrightarrow{\overline{\mathrm{c}}} A \mid B$$

We have explicitly unfolded B after the first step to make the interaction between $\overline{\mathrm{b}}$ 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 $\overline{\mathrm{b}}$ 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 $\mathsf{new}\, a.P$ for a process with a locally bound name a. Names bound with $\mathsf{new}\, a.P$ are subject to α-conversion (renaming of bound variables) as usual. In the example above, we would write

$$\mathsf{new}\, b.A \mid B.$$

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.

$$\begin{array}{rcl} A(a,b) & \stackrel{\mathrm{def}}{=} & \mathrm{a}.\overline{\mathrm{b}}.A\langle a,b\rangle \\ B(b,c) & \stackrel{\mathrm{def}}{=} & \mathrm{b}.\overline{\mathrm{c}}.B\langle b,c\rangle \end{array}$$

The process expression can now hygienically refer to locally bound names.

$$\mathsf{new}\, b.A\langle a,b\rangle \mid B\langle b,c\rangle$$

This leads to the following language of *concurrent process expressions*.

$$\begin{array}{rrcl} \text{Process Exps} & P & ::= & A\langle a_1,\ldots,a_n\rangle \mid N \mid (P_1 \mid P_2) \mid \mathsf{new}\, a.P \\ \text{Sums} & N & ::= & \alpha.P \mid N_1 + N_2 \mid 0 \\ \text{Action Prefix} & \alpha & ::= & \mathrm{a} \mid \overline{\mathrm{a}} \mid \tau \end{array}$$

In order to describe the possible transitions we use a *structural congruence*, written $P \equiv Q$ that allows us rearrange the pieces of a process expression in a meaning-preserving way. It is given by the following laws, which

can be applied anywhere in a process expression. We write FN for the free names in process expression.

$$
\begin{array}{l}
P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid 0 \equiv P \\
M + N \equiv N + M \qquad M + (N + N') \equiv (M + N) + N' \qquad M + 0 \equiv M \\
\mathsf{new}\, a.(P \mid Q) \equiv P \mid (\mathsf{new}\, a.Q) \qquad \text{provided } a \notin \mathrm{FN}(P) \\
\mathsf{new}\, a.P \equiv P \qquad \text{provided } a \notin \mathrm{FN}(P) \\
\mathsf{new}\, a.\mathsf{new}\, b.P \equiv \mathsf{new}\, b.\mathsf{new}\, a.P \\
A\langle b_1, \ldots, b_n \rangle \equiv \{b_1/a_1, \ldots, b_n/a_n\} P_A \qquad \text{provided } A(a_1, \ldots, a_n) \stackrel{\text{def}}{=} P_A
\end{array}
$$

Renaming of variables by new is implicit here. With this definition we can transform any process expression into a standard form

$$\mathsf{new}\, a_1 \ldots . \mathsf{new}\, a_n.(M_1 \mid \cdots \mid M_k)$$

where we write 0 if $k = 0$.

In order to define the operational semantics we take advantage of structural congruence to put the expressions that have to interact into proximity. In this semantics, all transitions are silent.

$$\frac{}{\tau.P + M \longrightarrow P}\ \text{Tau} \qquad \frac{}{(\mathrm{a}.P + M) \mid (\overline{\mathrm{a}}.Q + N) \longrightarrow P \mid Q}\ \text{React}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{Par} \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, a.P \longrightarrow \mathsf{new}\, a.P'}\ \text{New}$$

$$\frac{P \equiv Q \quad Q \longrightarrow Q' \quad Q' \equiv P'}{P \longrightarrow P'}\ \text{Struct}$$

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. As example for the above rules, consider the following process expression.

$$P = (\mathsf{new}\, a.((\mathrm{a}.Q_1 + \mathrm{b}.Q_2) \mid \overline{\mathrm{a}}.0)) \mid (\overline{\mathrm{b}}.R_1 + \overline{\mathrm{a}}.R_2)$$

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 transitions.

$$
\begin{array}{l}
P \longrightarrow (\mathsf{new}\, a.(Q_1 \mid 0)) \mid (\overline{\mathrm{b}}.R_1 + \overline{\mathrm{a}}.R_2) \\
P \longrightarrow (\mathsf{new}\, a.(Q_2 \mid \overline{\mathrm{a}}.0)) \mid R_1
\end{array}
$$

We next present an alternative semantics in which we do not need to resort to structural equivalence, except reassociating the terms in a sum. 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 $\overline{\mathrm{a}}$ and $\overline{\lambda}$ for $\overline{\mathrm{a}}$ or a, respectively.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum}_t \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{React}_t$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{L-Par}_t \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\ \text{R-Par}_t$$

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin \{\mathrm{a}, \overline{\mathrm{a}}\})}{\mathsf{new}\, a.P \xrightarrow{\alpha} \mathsf{new}\, a.P'}\ \text{Res}_t$$

$$\frac{\{b_1/a_1, \ldots, b_n/a_n\}P_A \xrightarrow{\alpha} P' \quad (A(a_1, \ldots, a_n) \stackrel{\text{def}}{=} P_A)}{A\langle b_1, \ldots, b_n\rangle \xrightarrow{\alpha} P'}\ \text{Ident}_t$$

As another example of this form of concurrent processes, consider two two-way transducers of identical structure.

$$A(a, a', b, b') \stackrel{\text{def}}{=} \mathrm{a}.\overline{\mathrm{b}}.A\langle a, a', b, b'\rangle + \mathrm{b}'.\overline{\mathrm{a}'}.A\langle a, a', b, b'\rangle$$

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 $\overline{\mathrm{a}}.P$, $\mathrm{c}.P'$, $\overline{\mathrm{c}'}.Q$, $\mathrm{a}'.Q'$

$$\begin{array}{l}
\overline{\mathrm{a}}.P \mid \mathrm{c}.P' \mid \overline{\mathrm{c}'}.Q \mid \mathrm{a}'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle) \\
\longrightarrow P \mid \mathrm{c}.P' \mid \overline{\mathrm{c}'}.Q \mid \mathrm{a}'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{\mathrm{b}}.A\langle a, a', b, b'\rangle \mid A\langle b, b', c, c'\rangle) \\
\longrightarrow P \mid \mathrm{c}.P' \mid Q \mid \mathrm{a}'.Q' \mid \mathsf{new}\, b.\mathsf{new}\, b'.(\overline{\mathrm{b}}.A\langle a, a', b, b'\rangle \mid \overline{\mathrm{b}'}.A\langle b, b', c, c'\rangle)
\end{array}$$

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.

$$
\begin{aligned}
&\overline{\mathrm{a}}.P \mid \mathrm{c}.P' \mid \overline{\mathrm{c}'}.Q \mid \mathrm{a}'.Q' \mid A\langle a, a', c, c'\rangle \\
&\longrightarrow P \mid \mathrm{c}.P' \mid \overline{\mathrm{c}'}.Q \mid \mathrm{a}'.Q' \mid \overline{\mathrm{c}}.A\langle a, a', c, c'\rangle \\
&\longrightarrow P \mid P' \mid \overline{\mathrm{c}'}.Q \mid \mathrm{a}'.Q' \mid A\langle a, a', c, c'\rangle \\
&\longrightarrow P \mid P' \mid Q \mid \mathrm{a}'.Q' \mid \overline{\mathrm{a}'}.A\langle a, a', c, c'\rangle \\
&\longrightarrow P \mid P' \mid Q \mid Q' \mid A\langle a, a', c, c'\rangle
\end{aligned}
$$

The reader should make sure to understand these transition and redesign the composed two-way buffer so that this deadlock situation cannot occur.

The two forms of semantics we have given are equivalent in the following way:

(i) If $P \longrightarrow P'$ then $P \stackrel{\tau}{\longrightarrow} P''$ and $P'' \equiv P'$ for some P''.

(ii) If $P \stackrel{\tau}{\longrightarrow} P'$ then $P \longrightarrow P'$.

These theorems are proving after appropriate generalization, by inductions over the given derivations. We do not give the proof here.

Supplementary Notes on The Pi-Calculus

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 24
November 19, 2002

In this lecture we first consider the question of observational equivalence for the calculus of concurrent, communicating processes. Then we extend the calculus to allow us communication to transmit values, which leads to the π-calculus.

Recall from the last lecture our notion of process expression, and in particular the unobservable (internal) action τ.

$$\begin{array}{rrcl} \text{Observable Action} & \lambda & ::= & \mathrm{a} \mid \overline{\mathrm{a}} \\ \text{Action} & \alpha & ::= & \lambda \mid \tau \\ \text{Process Exps} & P & ::= & A\langle a_1, \ldots, a_n\rangle \mid N \mid (P_1 \mid P_2) \mid \mathsf{new}\, a.P \\ \text{Sums} & N & ::= & \alpha.P \mid N_1 + N_2 \mid 0 \end{array}$$

The operational semantics with observable behavior is given by the judgment $P \xrightarrow{\alpha} P'$ which is defined by the following rules. Here we write $\overline{\lambda}$ for the opposite of λ, with the understanding that $\overline{\overline{\mathrm{a}}} = a$.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum}_t \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{React}_t$$

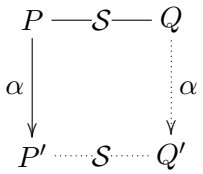
$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{L-Par}_t \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}\ \text{R-Par}_t$$

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin \{\mathrm{a}, \overline{\mathrm{a}}\})}{\mathsf{new}\, a.P \xrightarrow{\alpha} \mathsf{new}\, a.P'}\ \text{Res}_t$$

$$\frac{\{b_1/a_1, \ldots, b_n/a_n\}P_A \xrightarrow{\alpha} P' \quad (A(a_1, \ldots, a_n) \stackrel{\text{def}}{=} P_A)}{A\langle b_1, \ldots, b_n\rangle \xrightarrow{\alpha} P'}\ \text{Ident}_t$$

Also recall 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 $\overline{\mathrm{a}}$). 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

$$\begin{array}{lll} P \xrightarrow{\tau^*} P' & \text{iff} & P \xrightarrow{\tau} \cdots \xrightarrow{\tau} P' \\ P \xrightarrow{\tau^* \lambda \tau^*} P' & \text{iff} & P \xrightarrow{\tau^*} P_1 \xrightarrow{\lambda} P_2 \xrightarrow{\tau^*} P' \end{array}$$

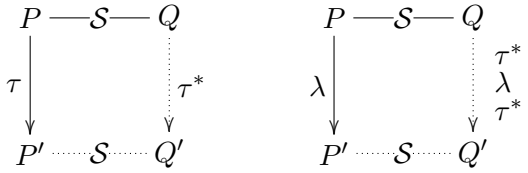
In particular, we always have $P \xrightarrow{\tau^*} P$. Then we say that $\mathcal{S}$ is a *weak simulation* if the following two conditions are satisfied:[1]

(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 $P' \mathcal{S} Q'$.

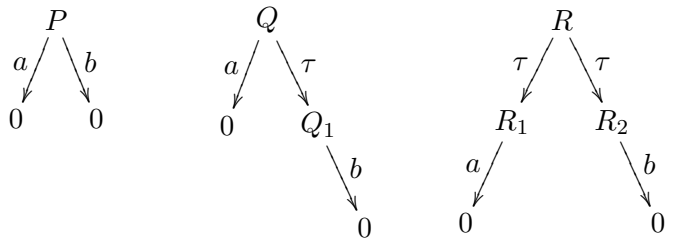
[1] This differs slightly, but I believe insignificantly from Milner's definition.

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

$$\begin{aligned} a.P &\approx \tau.a.P \\ a.P + \tau.a.P &\approx \tau.a.P \\ a.(b.P + \tau.c.Q) &\approx a.(b.P + \tau.c.Q) + \tau.c.Q \end{aligned}$$

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 desirect result: $Q_1 \approx Q_2$.

Next we will generalize the calculus of concurrent processes so that value 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.

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 equation by process replication $!P$ explained below.

Action prefixes	π	$::=$	$\mathrm{x}(y)$	receive y along x
		$\mid$	$\overline{\mathrm{x}}\langle y\rangle$	send y along x
		$\mid$	τ	unobservable action
Process exps	P	$::=$	$N \mid (P_1 \mid P_2) \mid \mathsf{new}\, a.P \mid\, !P$	
Sums	N	$::=$	$0 \mid N_1 + N_2 \mid \pi.P$	

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.

In examples $\pi.0$ is often abbreviated by π. Note that in a summand $\mathrm{x}(y).P$, y is a *bound variable* with scope P that stands for the value received along x. On the other hand, $\overline{\mathrm{x}}\langle y\rangle.P$ does not bind any variables.

Before presenting the transition semantics, we consider the following example.

$$P = ((\overline{\mathrm{x}}\langle y\rangle.0 + \mathrm{z}(w).\overline{\mathrm{w}}\langle y\rangle.0) \mid \mathrm{x}(u).\overline{\mathrm{u}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0)$$

The middle process can synchronize and communicate with either the first or the last one. Reaction with the first leads to

$$P_1 = (0 \mid \overline{\mathrm{y}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0) \equiv (\overline{\mathrm{y}}\langle v\rangle.0 \mid \overline{\mathrm{x}}\langle z\rangle.0)$$

which cannot transition further. Reaction with the seconds leads to

$$P_1' = ((\overline{\mathrm{x}}\langle y\rangle.0 + \mathrm{z}(w).\overline{\mathrm{w}}\langle y\rangle.0) \mid \overline{\mathrm{z}}\langle v\rangle.0 \mid 0)$$

which can step further to

$$P_2' = (\overline{\mathrm{v}}\langle y\rangle.0 \mid 0 \mid 0)$$

Next we show the reaction rules in a form which does not make an externally observable action explicit, and exploits structural congruence.

$$\frac{}{\tau.P + N \longrightarrow P}\ \text{Tau}$$

$$\frac{}{(\mathrm{a}(x).P + M) \mid (\overline{\mathrm{a}}\langle b\rangle.Q + N) \longrightarrow (\{b/x\}P) \mid Q}\ \text{React}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{Par} \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, x.P \longrightarrow \mathsf{new}\, x.P'}\ \text{Res}$$

$$\frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}\ \text{Struct}$$

Even though the syntax does not formally distinguish, we use x for binding occurrences of names (subject to silent renaming), and a and b for non-binding occurrences.

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.

$$\begin{aligned} C(x, \mathrm{get}, \mathrm{put}) \quad \stackrel{\mathrm{def}}{=} \quad & \overline{\mathrm{get}}\langle x\rangle.C\langle x, \mathrm{get}, \mathrm{put}\rangle.0 \\ & + \mathrm{put}(y).C\langle y, \mathrm{get}, \mathrm{put}\rangle.0 \end{aligned}$$

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.

$$!\,\mathrm{c}(x, \mathrm{get}, \mathrm{put}).(\overline{\mathrm{get}}\langle x\rangle.\overline{\mathrm{c}}\langle x, \mathrm{get}, \mathrm{put}\rangle.0 + \mathrm{put}(y).\overline{\mathrm{c}}\langle y, \mathrm{get}, \mathrm{put}\rangle.0)$$

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 $\overline{\mathrm{print}}\langle a\rangle.0$. We also consider 3 and 4 just as names here.

$$!C \mid \mathsf{new}\, g.\mathsf{new}\, p.\overline{\mathrm{c}}\langle 3, g, p\rangle.\overline{\mathrm{p}}\langle 4\rangle.\mathrm{g}(x).\overline{\mathrm{print}}\langle x\rangle.0$$

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.[2] The idea is to have a channel which sends successive numbers.

$$!\mathrm{count}(n, \mathrm{out}).\overline{\mathrm{out}}\langle n\rangle.\overline{\mathrm{count}}(n+1, \mathrm{out})$$

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 $(\mathrm{x} \bmod \mathrm{p} = 0)$ and its negation.

$$\begin{aligned}!\mathrm{filter}(p, \mathrm{in}, \mathrm{out}).\mathrm{in}(x).((\mathrm{x} \bmod \mathrm{p} = 0)().\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{out}\rangle.0 \\ + (\mathrm{x} \bmod \mathrm{p} \neq 0)().\overline{\mathrm{out}}(x).\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{out}\rangle.0)\end{aligned}$$

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).

$$\begin{aligned}!\mathrm{primes}(\mathrm{in}, \mathrm{out}).\mathrm{in}(p).\overline{\mathrm{out}}\langle p\rangle. \\ \mathsf{new}\, \mathrm{mid}.(\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \mid \overline{\mathrm{primes}}\langle \mathrm{mid}, \mathrm{out}\rangle.0)\end{aligned}$$

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 $\overline{\mathrm{out}}\langle p\rangle$ for each successive prime.

[2] This can also be coded in the π-calculus, but we prefer to avoid this complication here.

$$\mathsf{new}\,\mathrm{nats}.\overline{\mathrm{count}}\langle 2, \mathrm{nats}\rangle \mid \overline{\mathrm{primes}}\langle \mathrm{nats}, \mathrm{out}\rangle$$

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

$$\overline{\mathrm{a}}\langle b\rangle.P \quad \Longrightarrow \quad \tau.(\overline{\mathrm{a}}\langle b\rangle.0 \mid P)$$

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.

$$\begin{array}{l}!\mathrm{primes}(\mathrm{in}, \mathrm{out}).\mathrm{in}(p). \\ \quad \overline{\mathrm{out}}\langle p\rangle.0 \mid \mathsf{new}\,\mathrm{mid}.(\overline{\mathrm{filter}}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \;\mid\; \overline{\mathrm{primes}}\langle \mathrm{mid}, \mathrm{out}\rangle.0)\end{array}$$

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 neede 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 next lecture 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.

Supplementary Notes on Concurrent ML

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 25
November 21, 2002

In the last lecture we discussed the π-calculus, a minimal language with synchronous communication between concurrent processes. Mobility is modeled by allowing channels to transmit other channels, enabling dynamic reconfiguration of communication patterns.

Concurrent ML (CML) is an extension of Standard ML with concurrency primitives that heavily borrown from the π-calculus. In particular, 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:[1]

> 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
```

[1] See also `http://people.cs.uchicago.edu/~jhr/cml/`.

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.

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 on 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 head ch =
          let
            val p = CML.recv ch
          in
            CML.send (primes, p);
            head (filter (p, ch))
          end
    in
      CML.spawn (fn () => head (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*. Event 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.

Supplementary Notes on Environments

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 26
December 3, 2002

In the final two lectures of this course we go into slightly lower-level issues regarding the implementation of functional languages. As we will see, related issues arise in object-oriented languages.

This first observation about our semantic specifications is that most of them 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.

Because we are in a call-by-value language, environment η always 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.[1]

In order to concentrate on the essential issues with environments, we give here only a big-step operational semantics relating an expression to its final value. See [Ch. 11.2] for a version of the C-machine that maintains environments. We first give a big-step operational semantics using substitution. We concentrate on pairs and (non-recursive) functions; other constructs can be added but distract from the main issues.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\mathsf{pair}(e_1, e_2) \Downarrow \mathsf{pair}(v_1, v_2)}$$

$$\frac{e \Downarrow \mathsf{pair}(v_1, v_2)}{\mathsf{fst}(e) \Downarrow v_1} \qquad \frac{e \Downarrow \mathsf{pair}(v_1, v_2)}{\mathsf{snd}(e) \Downarrow v_2}$$

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e_3 \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_3 \Downarrow v}{\mathsf{apply}(e_1, e_2) \Downarrow v}$$

Next we try to add environments, being careful not to carry out substitutions, but just adding binding to the environments. The final two rules are actually incorrect, as we explain shortly.

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \mathsf{pair}(e_1, e_2) \Downarrow \mathsf{pair}(v_1, v_2)}$$

$$\frac{\eta \vdash e \Downarrow \mathsf{pair}(v_1, v_2)}{\eta \vdash \mathsf{fst}(e) \Downarrow v_1} \qquad \frac{\eta \vdash e \Downarrow \mathsf{pair}(v_1, v_2)}{\eta \vdash \mathsf{snd}(e) \Downarrow v_2}$$

$$\frac{}{\eta_1, x{=}v, \eta_2 \vdash x \Downarrow v}$$

$$\frac{}{\eta \vdash \lambda x.e \Downarrow \lambda x.e}\ ?? \qquad \frac{\eta \vdash e_1 \Downarrow \lambda x.e_3 \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta, x{=}v_2 \vdash e_3 \Downarrow v}{\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v}\ ??$$

If we now try to prove type preservation in the following form

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

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

we find that it is violated in the rule for λ-abstraction, since the value $\lambda x.e$ may have free variables referring to η. If we try to fix this problem by proving instead

If $\eta : \Gamma$ *and* $\Gamma \vdash e : \tau$ *and* $\eta \vdash e \Downarrow v$ *then* $\Gamma \vdash v : \tau$

the rule for λ-abstraction now works correctly, but the rule for application has a problem. This is because we eventually obtain from the induction hypothesis and multiple steps of reasoning that $\Gamma, x{:}\tau_2 \vdash v : \tau$, but we need that $\Gamma \vdash v : \tau$.

So neither of the two ideas works, and type preservation would be violated. In order to restore is, we need to pair up a value with its environment forming a *closure*. There are many strategies to make this efficient. For example, we could restrict the environment to those variables occurring free in the value, but we do not consider such refinements here. This means we have a new form of value, only used in the operational semantics, but not in the source expression.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \langle\!\langle \eta; \lambda x.e \rangle\!\rangle$$

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

$$\frac{}{\langle\!\langle \eta; \lambda x.e \rangle\!\rangle \ \mathsf{value}} \qquad \frac{\eta : \Gamma \quad \Gamma \vdash \lambda x.e : \tau}{\cdot \vdash \langle\!\langle \eta; \lambda x.e \rangle\!\rangle : \tau}$$

We now modify the incorrect rules by building and destructing closures instead.

$$\frac{}{\eta \vdash \lambda x.e \Downarrow \langle\!\langle \eta; \lambda x.e \rangle\!\rangle}$$

$$\frac{\eta \vdash e_1 \Downarrow \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta', x{=}v_2 \vdash e_3 \Downarrow v}{\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v}$$

Now it is easy to prove by induction over the structure of the evaluation that type preservation holds in the following form.

Theorem 1 (Type preservation with environments)
Assume $\eta : \Gamma$ *and* $\Gamma \vdash e : \tau$. *If* $\eta \vdash e \Downarrow v$ *then* $\cdot \vdash v : \tau$.

Proof: By induction on the derivation of $\eta \vdash e \Downarrow v$, applying inversion on the typing derivation in each case. We show the three critical cases.

Case: $\eta_1, x{=}v, \eta_2 \vdash x \Downarrow v$.

$\Gamma \vdash x : \tau$	Given
$\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$	By inversion
$\eta : \Gamma$	Given
$(\eta_1, x{=}v, \eta_2) : (\Gamma_1, x{:}\tau, \Gamma_2)$	Defns. of η and Γ
$\cdot \vdash v : \tau$	By inversion

Case: $\eta \vdash \lambda x.e \Downarrow \langle\!\langle \eta; \lambda x.e \rangle\!\rangle$.

$\Gamma \vdash \lambda x.e : \tau$	Given
$\eta : \Gamma$	Given
$\cdot \vdash \langle\!\langle \eta; \lambda x.e \rangle\!\rangle : \tau$	By rule

Case: $\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v$.

$\eta \vdash e_1 \Downarrow \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle$	Subderivation
$\eta \vdash e_2 \Downarrow v_2$	Subderivation
$\eta', x{=}v_2 \vdash e_3 \Downarrow v$	Subderivation
$\Gamma \vdash \mathsf{apply}(e_1, e_2) : \tau$	Given
$\Gamma \vdash e_1 : \tau_2 \to \tau$ and	
$\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\eta : \Gamma$	Given
$\cdot \vdash \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle : \tau_2 \to \tau$	By i.h.
$\eta' : \Gamma'$ and	
$\Gamma' \vdash \lambda x.e_3 : \tau_2 \to \tau$ for some Γ'	By inversion
$\Gamma', x{:}\tau_2 \vdash e_3 : \tau$	By inversion
$\cdot \vdash v_2 : \tau_2$	By i.h.
$(\eta', x{=}v_2) : (\Gamma', x{:}\tau_2)$	By rule
$\cdot \vdash v : \tau$	By i.h.

■

A big-step semantics is unsuitable for proving a progress theorem, so we will not do so here (see [Ch. 11.2]).

Type preservation tells us that the environment semantics we gave is sensible, but actually want to know more, namely that is is in an appropriate sense equivalent to the substitution semantics we gave earlier. This is another instance of a bisimulation theorem. It will be an instance of *weak* bisimulation because we do not care about the intermediate states of evaluation. In other words, we can only observe the final value returned by a computation. Even this we have to refine, as discussed in lecture 22.

There are three steps in proving a bisimulation theorem that shows the observational equivalence of two forms of operational semantics

1. Define the bisimulation relation.
2. Show that it is an observational equivalence.
3. Prove that it is a bisimulation.

We now go through these steps on the substitution and environment semantics.

Defining the bisimulation. Intuitively, the bisimulation substitutes out the environment. The main complication is that it must do this recursively, because the values in an environment can again contain closures and environments. The bisimulation decomposes into two judgments: one to relate expressions in an environment to closure-free expression, and one to relate values to values.

$\eta \vdash e \Longleftrightarrow e'$ e in environment η is related to e'
$v \longleftrightarrow v'$ v is related to v'

In order to describe the typing properties of the translation, we need to generalize environment to contain bindings $x{=}x'$. Typing for environments is then generalized as follows

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

$$\frac{\Gamma' \vdash \eta : \Gamma \quad \Gamma' \vdash x' : \tau}{\Gamma' \vdash (\eta, x{=}x') : (\Gamma, x{:}\tau)}$$

The typings are presupposed to be related as follows: if $\eta \vdash e \Longleftrightarrow e'$ then $\Gamma' \vdash \eta : \Gamma$ and $\Gamma \vdash e : \tau$ and $\Gamma' \vdash e' : \tau$ for some Γ and Γ'.

$$\frac{(x{=}v \in \eta) \quad v \longleftrightarrow v'}{\eta \vdash x \Longleftrightarrow v'} \qquad \frac{(x{=}x' \in \eta)}{\eta \vdash x \Longleftrightarrow x'}$$

$$\frac{\eta \vdash e_1 \Longleftrightarrow e_1' \quad \eta \vdash e_2 \Longleftrightarrow e_2'}{\eta \vdash \mathsf{apply}(e_1, e_2) \Longleftrightarrow \mathsf{apply}(e_1', e_2')}$$

$$\frac{\eta, x{=}x' \vdash e \Longleftrightarrow e'}{\eta \vdash \lambda x.e \Longleftrightarrow \lambda x'.e'}$$

$$\frac{\eta' \vdash \lambda x.e \Longleftrightarrow v'}{\langle\langle \eta'; \lambda x.e \rangle\rangle \longleftrightarrow v'}$$

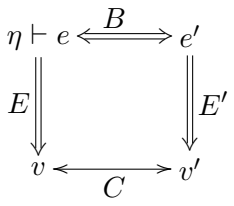
Observational Equivalence. Since we have simplified our language to just contain functions, the observational equivalance is trivialized. However, if we add, for example, integers and primitive operations, then we would have the rules

$$\frac{\eta \vdash e_1 \Longleftrightarrow e_1' \quad \eta \vdash e_2 \Longleftrightarrow e_2'}{\eta \vdash o(e_1, e_2) \Longleftrightarrow o(e_1', e_2')}$$

$$\frac{}{\mathsf{int}(n) \longleftrightarrow \mathsf{int}(n)}$$

and it is indeed the case that $\longleftrightarrow$ coincides with equality on the observable type int.

Proving the bisimulation. Bisimulation in this case can be discussed using the following diagram.



We need to show two properties:

1. If B and E are given, then v', E', and C exist, and
2. if B and E' are given, then v, E, and C exist.

Fortunately, neither of these direction is difficult. We do, however, need a substitution property.

Theorem 2 (Substitution for bisimulation)
If $v \longleftrightarrow v'$ *and* $\eta_1, x{=}x', \eta_2 \vdash e \Longleftrightarrow e'$ *then* $\eta_1, x{=}v, \eta_2 \vdash e \Longleftrightarrow \{v'/x'\}e'$.

Proof: By induction on the derivation of $\eta_1, x{=}x', \eta_2 \vdash e \Longleftrightarrow e'$. ■

For a more general language, we also need the easy property that $e \Downarrow e$ iff e value in the substitution semantics.

Theorem 3 (Simulation$_1$)
If $\eta \vdash e \Longleftrightarrow e'$ *and* $\eta \vdash e \Downarrow v$ *then there exists a* v' *such that* $e' \Downarrow v'$ *and* $v \longleftrightarrow v'$.

Proof: By induction on the derivation of $\eta \vdash e \Downarrow v$ and inversion on $\eta \vdash e \Longleftrightarrow e'$ in each case. ■

Theorem 4 (Simulation$_2$)
If $\eta \vdash e \Longleftrightarrow e'$ and $e' \Downarrow v'$ then there exists a v such that $\eta \vdash e \Downarrow v$ and $v \longleftrightarrow v'$.

Proof: By induction on the derivation of $e' \Downarrow v'$ and inversion on $\eta \vdash e \Longleftrightarrow e'$ in each case. ■

There is a compile-time analogue to the closures that are generated in our operational semantics at run-time. This is the so-called *closure conversion*. To see the need for that, consider the simple program (shown in SML syntax)

```
let val x = 1
    val y = 2
in fn w => x + w + 1 end
```

How do we compile the function `fn w => x + w + 1`? The difficulty here is the reference to variable `x` defined in the ambient environment.

The solution is to close the code by abstracting over an environment, and pairing it up with the environment. This way we obtain

```
let val x = 1
    val y = 2
in (fn env => fn w => (#x env) + w + 1, {x = x}) end
```

If this transformation is carried out systematically, all functions are closed and can be compiled to a piece of each. Each of them expect and environment as an additional argument. This environment contains only the bindings of variables that actually occur free in the body of the function. An application of the function now also applies the function to the environment. For example,

```
let val x = 1
    val y = 2
    val f = fn w => x + w + 1
in f 3 end
```

is translated to

```
let val x = 1
    val y = 2
    val f = (fn env => fn w => (#x env) + w + 1,
             {x = x})
in (#1 f) (#2 f) 3  end
```

The problem with this transformation is that its target is generally not well typed. This is because functions with different sets of free variables will sometimes have different type. For example, the code

```
let val x = 1
 in if true then fn w => w + x
    else fn w => w + 2
end : int -> int
```

becomes

```
let val x = 1
 in if true
    then (fn env => fn w => w + (#x env), {x = x})
    else (fn env => fn w => w + 2, {})
end
```

which is not well-typed because the two branches of the conditional have different type: the first has type `({x:int} -> int -> int) * {x:int}` and the second has type `({} -> int -> int) * {}`. We can repair the situation by using existential types. Since SML does not have first-class existential type, we just use the syntax `pack[t](e)` for $\mathsf{pack}(\tau, e)$. Then the example above can be written as

```
let val x = 1
 in if true
    then pack [{x:int}]
         (fn env => fn w => w + (#x env), {x = x})
    else pack [{}] (fn env => fn w => w + 2, {})
end
```

which now has type

$$\exists t.(t \to \mathsf{int} \to \mathsf{int}) \times t.$$

An application of a function before the translation is now translated to use `open`. For example,

```
let val x = 1
    val y = 2
    val f = pack [{x:int}]
            (fn env => fn w => (#x env) + w + 1,
             {x = x})
in open f as 'a => g => (#1 g) (#2 g) 3  end
```

where we wrote `open e1 as t => g => e2` for $\mathsf{open}(e_1, t.g.e_2)$.

The above discussion can be summarized by *closures have existential type*. When we apply this idea to objects in our encoding, we find that objects in a functional language are also become closures and therefore have existential type. Reconsider the simple example of a counter.

```
Counter = {get:1 -> int, inc:1 -> 1};
c : Counter =
    let x = ref 1
     in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

After closure conversion, we obtain

```
c : Counter =
    let x = ref 1
     in
        pack [{x:int}]
        (λr.
         {get = λ_:1. !r.x,
         inc = λ_:1. r.x := !r.x+1},
         {x=x})
    end;
```

which has type

$$\exists t.(t \to \{\mathsf{get} : 1 \to \mathsf{int}, \mathsf{inc} : 1 \to 1) \times t$$

Here, the existential type hides the private fields of the record. This justifies the slogan that objects have existential type.

Supplementary Notes on Storage Management

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 27
December 5, 2002

In this lecture we discuss issues of storage managment and garbage collection. The lecture follows [Ch. 31] rather closely, so we concentrate on what is slightly different here, namely the presentation of bisimulation.

In order to talk about garbage collection, we need to formalize the distinction between *small values* and *large values*, where small values can be part of the stack, while large values must be allocated in the heap. For the purpose of this lecture, small values are either integers, booleans, or "pointers" to large values. Functions are large values, as are pairs. We use the judgments v svalue and v lvalue for small and large values, respectively. Pointers are represented by a new kind of value l, standing for locations in memory. Unlike with mutable storage, such locations cannot be changed, but they can be allocated (implicitly) and deallocated (by garbage collection).

$$\frac{}{\mathsf{int}(n)\ \mathsf{svalue}} \qquad \frac{}{l\ \mathsf{svalue}}$$

$$\frac{v_1\ \mathsf{svalue} \quad v_2\ \mathsf{svalue}}{\mathsf{pair}(v_1, v_2)\ \mathsf{lvalue}} \qquad \frac{}{\lambda x.e\ \mathsf{lvalue}}$$

Note that the components of a pair are small values. This means they must either be integers, booleans, or again pointers.

Heaps are simply locations together with their (immutable) values.

$$\text{Heaps} \quad H \quad ::= \quad \cdot \mid H, l{=}v$$

As usual, all location l must be distinct. Furthermore, values stored in the heap must be large values, that is, if $l{=}v$ is part of the heap, then v lvalue.

We describe a heap-based operational semantics using the A-machine, which is an extension of the C-machine to account for the heap. Recall:

$$\begin{array}{rrcl} \text{Stacks} & K & ::= & \bullet \mid K \triangleright \mathsf{apply}(\Box, e_2) \mid K \triangleright \mathsf{apply}(v_1, \Box) \\ & & \mid & K \triangleright \mathsf{pair}(\Box, e_2) \mid K \triangleright \mathsf{pair}(v_1, \Box) \mid K \triangleright \mathsf{fst}(\Box) \mid K \triangleright \mathsf{snd}(\Box) \end{array}$$

$$\begin{array}{rrcl} \text{States} & s & ::= & K > e \mid K < v \end{array}$$

We restrict stacks and states to contain only small values. If an expression e is stored on the stack or in the process of being evaluated it has not yet been turned into a value and therefore does not have to satisfy this criterion. We show only a few rules which formalize this intuition. We ignore issues of typing, since they are largely orthogonal and basically unchanged from the typing of the C-machine.

$$\frac{K\ \mathsf{stack} \quad e\ \mathsf{exp}}{K > e\ \mathsf{state}} \qquad \frac{K\ \mathsf{stack} \quad v\ \mathsf{svalue}}{K > v\ \mathsf{state}}$$

$$\frac{K\ \mathsf{stack} \quad e_2\ \mathsf{exp}}{K \triangleright \mathsf{apply}(\Box, e_2)\ \mathsf{stack}} \qquad \frac{K\ \mathsf{stack} \quad v_1\ \mathsf{svalue}}{K \triangleright \mathsf{apply}(v_1, \Box)\ \mathsf{stack}}$$

$$\frac{}{\cdot\ \mathsf{heap}} \qquad \frac{H\ \mathsf{heap} \quad v\ \mathsf{lvalue} \quad (l \notin \mathrm{dom}(H))}{H, l{=}v\ \mathsf{heap}}$$

A machine state is now extended by a heap, written as $H; s$, where s is either $K > e$ or $K < v$. There are several invariants we will want to maintain of machine states. For example, it should be *self-contained*. If we denote the location defined by a heap with $\mathrm{dom}(H)$ and the free locations in a term (that is, expression, value, stack, or values defined in a heap) by $\mathrm{FL}(\_)$, the $H; s$ is self-contained if $\mathrm{FL}(H) \cup \mathrm{FL}(s) \subseteq \mathrm{dom}(H)$. For other invariants, see [Ch. 31].

The transitions of the A-machine can now be developed in analogy with the C-machine, keeping in mind that we need to maintain the distinction between small and large values.

$$\begin{array}{lcll} H; K > \mathsf{pair}(e_1, e_2) & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{pair}(\Box, e_2) > e_1 & \\ H; K \triangleright \mathsf{pair}(\Box, e_2) < v_1 & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{pair}(v_1, \Box) > e_2 & \\ H; K \triangleright \mathsf{pair}(v_1, \Box) < v_2 & \mapsto_{\mathsf{a}} & H, l{=}\mathsf{pair}(v_1, v_2); K < l & (l \notin \mathrm{dom}(H)) \\ H; K > \mathsf{fst}(e) & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{fst}(\Box) > e & \\ H; K \triangleright \mathsf{fst}(\Box) < l & \mapsto_{\mathsf{a}} & H; K < v_1 & (l{=}\mathsf{pair}(v_1, v_2) \in H) \\ H; K > \mathsf{snd}(e) & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{snd}(\Box) > e & \\ H; K \triangleright \mathsf{snd}(\Box) < l & \mapsto_{\mathsf{a}} & H; K < v_2 & (l{=}\mathsf{pair}(v_1, v_2) \in H) \end{array}$$

It is easy to verify inductively that the value size invariants for heaps and stacks are preserved by these rules. We finish with the rules for functions.

$$
\begin{array}{lcll}
H; K > \mathsf{apply}(e_1, e_2) & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{apply}(\Box, e_2) > e_1 & \\
H; K \triangleright \mathsf{apply}(\Box, e_2) < v_1 & \mapsto_{\mathsf{a}} & H; K \triangleright \mathsf{apply}(v_1, \Box) > e_2 & \\
H; K \triangleright \mathsf{apply}(l_1, \Box) < v_2 & \mapsto_{\mathsf{a}} & H; K > \{v_2/x\}e_1 & (l_1{=}\lambda x.e_1 \in H) \\
H; K > \lambda x.e & \mapsto_{\mathsf{a}} & H, l{=}\lambda x.e; K < l & (l \notin \mathrm{dom}(H))
\end{array}
$$

Next we would like to show the correctness of the A-machine when compared to the C-machine. Interestingly, this becomes a *strong* bisimulation theorem: the two machines execute in lock-step. This requires that we set up a bisimulation relation. Following some of the prior examples we have seen, this amounts to substituting values for location labels l. Also as before, this has to be done recursively, because the values v can again contain references to other values, and so on. The inductive definition of the bisimulation as a judgment is not difficult, but somewhat tedious, so we only show a few cases. We have the judgments

$$
\begin{array}{l}
H; K \sim K' \\
H; e \sim e' \\
H; v \sim v' \\
H; s \sim s'
\end{array}
$$

defined by the following rules:

$$
\frac{H; v \sim v' \quad (l{=}v \in H)}{H; l \sim v'} \qquad \frac{H; K \sim K' \quad H; e \sim e'}{H; K > e \sim K' > e'} \qquad \frac{H; K \sim K' \quad H; v \sim v'}{H; K < v \sim K' < v'}
$$

$$
\frac{}{H; \bullet \sim \bullet} \qquad \frac{H; K \sim K' \quad H; e_2 \sim e_2'}{H; K \triangleright \mathsf{apply}(\Box, e_2) \sim K' \triangleright \mathsf{apply}(\Box, e_2')}
$$

$$
\frac{H; K \sim K' \quad H; v_1 \sim v_1'}{H; K \triangleright \mathsf{apply}(v_1, \Box) \sim K' \triangleright \mathsf{apply}(v_1', \Box)}
$$

$$
\frac{H; e_1 \sim e_1' \quad H; e_2 \sim e_2'}{H; \mathsf{apply}(e_1, e_2) \sim \mathsf{apply}(e_1', e_2')} \qquad \frac{H; e \sim e'}{H; \lambda x.e \sim \lambda x.e'} \qquad \frac{}{H; x \sim x}
$$

These rules are extended to handle pairs and states using straightforward congruence rules for all constructs. The fact this works essentially works like a congruence yields the following lemma. We take exchange for granted: the order of the locations in the heap is irrelevant. We use O and O' to stand for stacks, expressions, values, or states.

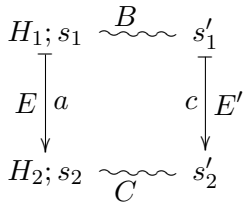
Lemma 1 (Weakening and Substitution)

(i) If $H; O \sim O'$ *then* $H, H'; O \sim O'$

(ii) If $H; v \sim v'$ *and* $H; O \sim O'$ *then* $H; \{v/x\}O \sim \{v'/x\}O'$

Proof: By induction on the structure of the given derivation relating O and O'. ■

Now we can prove strong bisimulation according to the following diagram:



We have to show (1) that if B and E are given, then s'_2, E', and C exist, and (2) that if B and E' are given, then H_2, s_2, E, and C exist.

Theorem 2 (Strong Bisimulation for A- and C-machine)

(i) If $H_1; s_1 \sim s'_1$ *and* $H_1; s_1 \mapsto_{\mathsf{a}} H_2; s_2$ *then there is an* s'_2 *such that* $s'_1 \mapsto_{\mathsf{c}} s'_2$ *and* $H_2; s_2 \sim s'_2$.

(ii) If $H_1; s_1 \sim s'_1$ *and* $s'_1 \mapsto_{\mathsf{c}} s'_2$ *then there is an* H_2 *and* s_2 *such that* $H_1; s_1 \mapsto_{\mathsf{a}} H_2; s_2$ *and* $H_2; s_2 \sim s'_2$.

Proof: In direction (1) we examine the cases for E, applying inversion to B to construct s'_2 and E'. In direction (2) we examine the cases for E', applying inversion to B to construction H_2, s_2 and C. ■

On observable values (i.e. integers, pairs of integers, etc.) the simulation yields the right translation, as can readily be verified.

Once we have heaps, we can formulate garbage collection as a way to trace through the heap and copying all locations accessible in a state. In some ways this inverts the weakening lemma into a *strengthening* property, where we remove part of the heap H' that is not referred to in the stack or expression. Garbage collection admits only weak bisimulation, since the steps of garbage collection are not accounted for in the C-machine. Please see [Ch. 31] for further details and discussion of garbage collection.