Scribe: Kamal Aboul-Hosn and Ganesh Ramanarayanan Lecturer: Radu Rugina

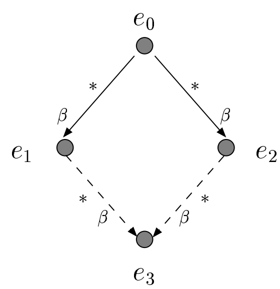
1 Confluence Property

Last lecture we've seen many different evaluation orders for lambda expressions, including CBV, CBN, Normal Order, Applicative Order, and Full Beta Reduction. A natural question that arises is whether applying different evaluation orders may lead to different results. The answer is no, and is due to a fundamental result in Lambda Calculus, known as the Church-Rosser Theorem.

Theorem 1 (Church-Rosser) *The β relation satisfies the confluence (diamond) property, i.e.,*

$$\forall e_0, e_1, e_2 \ . \ (e_0 \rightarrow^*_\beta e_1 \ \wedge \ e_0 \rightarrow^*_\beta e_2) \ \Rightarrow \ (\exists e_3 \ . \ e_1 \rightarrow^*_\beta e_3 \ \wedge \ e_2 \rightarrow^*_\beta e_3)$$

This theorem states that even though different evaluation orders for an expression e_0 may lead to different expressions e_1 and e_2, we can always "continue" these evaluations and get the same final expression e_3. Graphically, this is represented as follows:



Corollary 1 *If it exists, the β normal form is unique.*

This corollary immediately follows from the theorem. Consider an expression e an let e_1 and e_2 be normal forms of e. We show that $e_1 = e_2$. By the Church-Rosser theorem, there exists some expression e_3 such that $e_0 \rightarrow^*_\beta e_3$ and $e_1 \rightarrow^*_\beta e_3$. Since e_0 and e_1 are in normal form, they contain no redexes. Hence their evaluations to e_3 have zero steps, so $e_1 = e_3 = e_2$.

It turns out that similar results also hold for the $\beta\eta$ relation and for the η relation.

Theorem 2 *Each of the relations $\beta\eta$ and η satisfy the confluence property.*

Corollary 2 *If they exist, $\beta\eta$-nf and η-nf are unique.*

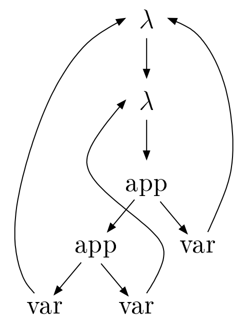
2 Nameless Representations

We've seen that α-reduction just renames bound variables in abstractions; it doesn't change the structure of the expression. We should be able to construct *nameless* representations, which precisely capture the structure of lambda expressions, without using variable names. There are two possibilities: use diagrams or use a representation based on integer indices.

- *Representation using diagrams.* The idea is to represent relations between variables and their enclosing λ's as edges in a diagram. We can take a lambda expression, replace each variable name with a * symbol, and then draw edges around this expression to indicate, for each occurrence of *, the enclosing $\lambda *$ to which it corresponds. For instance, the nameless diagram representation of $\lambda x. \lambda y. (xy)x$ is:

$$\lambda * . \ \lambda * . \ (* \ *) \ *$$

Alternatively, we can draw the parse tree of the expression, using λ nodes for abstractions, *app* nodes for applications, and *var* nodes for variables. Then we add explicit edges between *var* nodes and λ nodes. For instance, the parse tree representation of $\lambda x.\, \lambda y.\, (xy)x$ is:



- *Representation using indices.* The idea is to use integer indices instead of variables. Each variable occurrence is labeled with index n if that variable is bound to the n-th enclosing λ. These numbers are called *De Bruijn indices*. For instance, the DeBruijn representation of $\lambda x.\, (\lambda y.\, (xy))x$ is:

$$\lambda\,.\,(\lambda\,.\,(1\ 0))\ 0$$

It is interesting to note that different occurrences of the same variable can get different indices in this representation, even if they are bound to the same λ. In this example, the two occurrences of x refer to the argument of the same abstraction. But the first occurrence has index 1 because it appears in nested abstractions, at the first level of nesting. In contrast, the second x has index 0 because it directly occurs in the body of its abstraction (zero-th level of nesting). Here are two other examples:

$$\begin{aligned} \lambda x.\, \lambda y.\, (xy)x &\equiv \lambda\,.\,\lambda\,.\,(1\ 0)\ 1 \\ Y_{CBV} = \lambda f.\, (\lambda x.\, f(\lambda y.\, xxy))(\lambda x.\, f(\lambda y.\, xxy)) &\equiv \lambda\,.\,(\lambda\,.\,1(\lambda\,.\,1\ 1\ 0))(\lambda\,.\,1(\lambda\,.\,1\ 1\ 0)) \end{aligned}$$

In the above representations we have assumed expressions with no free variables, i.e., combinators. For expressions with free variables, we need a naming context for the unbound variables. Another issue is how to define substitutions for such nameless representations. Defining substitution is still not straightforward, but we will not go into the details of how it works. We will instead continue to use the name-based representation of lambda terms and our previously defined substitution operation.

3 Checkpoint

We will move now to a next part of the course. But before we do that, I will step back for a moment and remind you what the big picture is, because you may have been caught in the details of our lectures. The course is about semantics and formal methods for programming languages. Semantics give us a set of formal tools that allow us:

1. To give precise specifications of programming languages;
2. To formally check program equivalence, termination, determinism;
3. To formally prove other program properties;

Learning about semantics also gives us more insight about language design, language expressiveness, and about how powerful can languages be. So far, we have considered three languages: IMP (imperative language), REC (applicative language, with recursive functions), and λ-calculus (simple, but powerful language with higher-order functions).

There are still a number of features that we haven't discussed yet. These include scoping, unstructured control flow and exceptions, dynamic allocation, and types. We will next discuss such language constructs and show their operational or denotational semantics.

4 The Functional Language F

We will first start with a base functional language called F that combines the features of the languages discussed so far. Later, we will add a number of language extensions to this base language. Hence, we use F as a vehicle for exploring other language constructs.

The functional language F borrows the following constructs from the languages discussed so far: truth values (IMP), integers and arithmetic (IMP, REC), conditional expressions (REC), functions and higher-order functions (λ-calculus), recursive functions (REC), plus two other constructs: pairs and let expressions. The syntax of F is:

$$
\begin{array}{rcl}
e & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \\
& & e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_0 \mid e_0 \leq e_1 \mid e_0 = e_1 \mid \\
& & \mathsf{if}\, e_0 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 \mid x \mid \lambda x.\, e \mid e_0 e_1 \mid \mathsf{rec}\ y\ (\lambda x.\ e) \mid \\
& & \langle e_0, e_1 \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1
\end{array}
$$

4.1 Small-Step Operational Semantics

We want to define F as a call-by-value language, with left-to-right evaluation of expressions. We will express these properties in the semantics of the language. Note that the semantics must also require operands of appropriate values in each expression. For instance the first operand in an application must be an abstraction; it cannot be a pair, an integer, or a truth value. Such cases represent *errors* and the semantics should forbid them to occur during the execution.

We express the operational semantics as a relation of the form $e \rightarrow_1 e'$, which evaluates e to e' in one step. The operational semantics for the evaluation of arithmetic and boolean expressions is:

$$
\frac{e_0 \rightarrow e_0'}{e_0 \diamond e_1 \rightarrow e_0' \diamond e_1} \qquad \frac{e_1 \rightarrow e_1'}{v \diamond e_1 \rightarrow v \diamond e_1'} \qquad \frac{}{v_0 \diamond v_1 \rightarrow v}\ v = v_0 \diamond v_1
$$

where

$$
\begin{array}{rcl}
\diamond & \in & \{+, -, *\} \\
v & ::= & n \mid \mathsf{true} \mid \mathsf{false}
\end{array}
$$

Note that values v can be either integers or truth values. The semantics requires that the operands of arithmetic or boolean expressions mush be such values. This forbids pairs or abstractions to be operands of such expressions. For such error cases, there will be no rule that applies and the execution gets into a *stuck configuration.*

The semantics of applications and abstractions is:

$$
\frac{e_0 \rightarrow e_0'}{e_0\ e_1 \rightarrow e_0'\ e_1} \qquad \frac{e_1 \rightarrow e_1'}{(\lambda x.e)\ e_1 \rightarrow (\lambda x.e)\ e_1'} \qquad \frac{}{(\lambda x.e)\ v \rightarrow e[v/x]}
$$

For pairs we have:

$$
\frac{e_0 \rightarrow e_0'}{\langle e_0, e_1 \rangle \rightarrow \langle e_0', e_1 \rangle} \qquad \frac{e_1 \rightarrow e_1'}{\langle v, e_1 \rangle \rightarrow \langle v, e_1' \rangle}
$$

$$
\frac{}{\mathsf{fst}\ \langle v_1, v_2 \rangle \rightarrow v_1} \qquad \frac{}{\mathsf{snd}\ \langle v_1, v_2 \rangle \rightarrow v_2}
$$

For let constructs:

$$
\frac{e_0 \rightarrow e_0'}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \rightarrow \mathsf{let}\ x = e_0'\ \mathsf{in}\ e_1} \qquad \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e_1 \rightarrow e_1[x \rightarrow v]}
$$

In all of the above rules, values are integers, truth values, abstractions, or pairs of values:

$$
v \ ::= \ n \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\, e \mid \langle v_1, v_2 \rangle
$$

These values enforce that operands agree with the values that expressions expect. Any error translates into a stuck configuration, where no semantic rule applies.

I claim that the semantics of rec constructs should be the following:

$$\frac{}{\mathsf{rec}\ y\ (\lambda x.\ e) \to (\lambda x. e)[y \to \mathsf{rec}\ y\ (\lambda x.\ e)]}$$

Here's the intuition behind this rule. The recursive function defined by the rec construct is the fixed point of a higher-order function g:

$$\begin{array}{rcl} g & = & \lambda y. \lambda x. e \\ \mathsf{rec}\ y\ (\lambda x.\ e) & = & \mathit{fix}\ (g) \end{array}$$

Therefore:

$$\begin{array}{rcl} \mathsf{rec}\ y\ (\lambda x.\ e) & = & g\ (\mathsf{rec}\ y\ (\lambda x.\ e))) \\ & = & (\lambda y. \lambda x. e)\ (\mathsf{rec}\ y\ (\lambda x.\ e)) \\ & = & (\lambda x. e)[y \to \mathsf{rec}\ y\ (\lambda x.\ e)] \end{array}$$