2024 OPLSS: Foundations of Programming Languages - Lecture 3

Lecturer: Paul Downen
Note Takers: Aaryan Patel, Adriano Corbelino II, Anders Thuné, Thomas Porter

June 2024

1 Multi-step Reduction

One approach to reason about program equivalence is the operational semantics. When comparing two programs, we can use a multi-step (reflexive transitive closure of $\mapsto_\beta$) reduction on both sides and then compare if they are equal. Those reductions are defined by the following inference rules:

$$\frac{M \mapsto_\beta M'}{E[M] \mapsto_\beta E[M']}\ \textsc{Compat}$$

$$\frac{M \mapsto_\beta M'}{M \mapsto\!\!\!\!\mapsto_\beta M'}\ \textsc{Incl} \qquad \frac{}{M \mapsto\!\!\!\!\mapsto_\beta M}\ \textsc{Refl} \qquad \frac{M \mapsto\!\!\!\!\mapsto_\beta M' \qquad M' \mapsto\!\!\!\!\mapsto_\beta M''}{M \mapsto\!\!\!\!\mapsto_\beta M''}\ \textsc{Trans}$$

The Inclusion (Incl) rule tells us that if we can reduce a term M to a term M' in a single step, then we can reduce M to M' in multiple steps. The Reflexivity (Refl) rule asserts that a term can reduce to itself. The Transitivity (Trans) rule defines that if we can reduce, in multiple steps, a term M to another term M' that can be further reduced to M'', then we can reduce M to M'' in multiple steps. The Compatibility (Compat) rule enables us to reduce according to our evaluation contexts.

2 Observational Equivalence (Contextual Equivalence)

2.1 Motivation and Set-up

Let's think about what it means for two programs to be equal. We will use the following auxiliary functions.

$not = \lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathit{false}\ \mathsf{else}\ \mathit{true}$
$and = \lambda y.\ \lambda x.\ \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ \mathit{false}$

Now, consider the following equality statements between programs.

- $\mathit{true} \stackrel{?}{=} \mathit{false}$
  These two programs are not equivalent as they terminate with different values.
- $\mathit{false} \stackrel{?}{=} \mathit{not\ true}$
  Following the operational semantics, these terms reduce to the same value, so we can consider them equivalent.
- $\mathit{false} \stackrel{?}{=} \mathit{and\ false\ x}$
  Again, the operational semantics reduce both terms to *false* (even though the right-hand-side is open).

- $false \stackrel{?}{=} and\ x\ false$
- $\lambda x.\ and\ false\ x \stackrel{?}{=} \lambda x.\ false$
- $\lambda x.\ not\ (not\ x) \stackrel{?}{=} \lambda x.\ x$

Notice that operational semantics aren't enough for the last three programs to establish equivalence. Yet each equation holds in some intuitive sense. For any instantiation of x, these equations hold. We will consider a more powerful notion of program equivalence to capture this intuition.

2.2 Definition

To reason about equivalent terms, we will use a notion of observational (or contextual) equivalence. Intuitively, two terms are observationally equal if they are interchangeable in any program context. Before we proceed, we need general program contexts $C \in Contexts$, which are like the evaluation contexts E, except that the hole can occur in any possible subterm.

Exercise 1: Define the contexts for the simply typed lambda calculus utilizing BNF notation.

We say that two terms M and N are *observationally equivalent*,

$$M \approx N, \text{ if and only if } M \preccurlyeq N \text{ and } N \preccurlyeq M.$$

The relation $M \preccurlyeq N$ is read "M approximates N", and captures a one-sided version of observational equivalence. We say that $M \preccurlyeq N$ if and only if, for all $C \in Contexts$ and $v \in Values$, if

$$FV(C[M]) = \emptyset = FV(C[N]) \quad \text{and} \quad C[M] \mapsto\!\!\!\!\rightarrow_\beta v$$

then there exists a value w such that

$$C[N] \mapsto\!\!\!\!\rightarrow_\beta w \quad \text{and} \quad w \sim v.$$

This relies on a final notion of similar values, $v \sim w$, defined inductively as follows.

$$\frac{}{true \sim true} \qquad \frac{}{false \sim false} \qquad \frac{}{\lambda x.\ M \sim \lambda y.\ N}$$

In other words, M approximates N if evaluating M in any (closing) context gives a similar result to evaluating N in that same context. The definition of $\sim$ for the lambda case may seem lax since it considers any two lambdas similar regardless of their bodies. However, this does not pose a problem for our approximation relation, since we consider *all* contexts C, including application to different arguments which would cause different behavior.

Example 2.1. *We have the following statements about approximation, for nonterminating* Ω*:*

$$\Omega \preccurlyeq true$$
$$true \not\preccurlyeq \Omega.$$

Since Ω never reduces to a value we can vacuously infer that it approximates to *true*. However, the converse is not valid.

Observational equality is an extremely informative property and is the "gold standard" notion of equality between programs. The issue is that due to the quantification over the broad class of contexts, it is difficult to prove in general. Next, we will try to find an approximate notion of program equivalence that is more syntactic and amenable to proof.

2.3 General Reduction and β-equivalence

The operational semantics only lets us step in a restricted set of evaluation contexts E. In contrast, we can define a general reduction of a term that is compatible with all contexts $C \in \mathit{Contexts}$. We define the single- and multi-step relations as follows, and use them to define a general syntactic notion of equality on terms.

- $\to_\beta$ is the *compatible* closure of $\mapsto_\beta$. In other words, it is the closure with respect to the following rule:

$$\frac{M \mapsto_\beta M'}{C[M] \to_\beta C[M']}$$

.

- $\twoheadrightarrow_\beta$ is the reflexive, transitive closure of $\to_\beta$.
- $=_\beta$ is the symmetric, transitive, reflexive closure of $\to_\beta$.

We would like to show that β-equivalence $=_\beta$ implies observational equality, thus giving an easy method for proving the latter in many cases.

We proceed by showing two properties, which together will be enough to prove this fact, namely *confluence* and *standardization*.

3 Confluence

Confluence is a property related to determinism, stating that whenever a term can be reduced in two different ways, there is a common term "joining the paths". There are different related notions of confluence, of various strengths. These definitions are generic for a given relation $\to$, with $\twoheadrightarrow$ denoting the reflexive, transitive closure of $\to$ and $=$ denoting the symmetric, reflexive, transitive closure of $\to$.

Definition 3.1 ((Strong) Confluence)**.** *A reduction relation is strongly confluent if every multi-step split $M_1 \twoheadleftarrow M \twoheadrightarrow M_2$ can always be joined by multiple steps $M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$.*

The following is a simple consequence of the definition which we will use.

Theorem 3.1. *For a (strongly) confluent $\to$, if $M = N$ then $M \twoheadrightarrow \cdot \twoheadleftarrow N$.*

We would like to show that β-reduction is confluent, but this is not straightforward. Let us consider some approaches.

Definition 3.2 (Diamond Property)**.** *A reduction relation has the diamond property if every one step split $M_1 \leftarrow M \to M_2$ can always be joined by a single step $M_1 \to M' \leftarrow M_2$.*

The diamond property clearly implies the confluence property. However, it does not hold for $\to_\beta$. For example, consider $(\lambda x.\ \mathsf{if}\ z\ \mathsf{then}\ x\ \mathsf{else}\ x)\ ((\lambda y.\ y)\ \mathit{false})$. It can take a single step to $(\lambda x.\ \mathsf{if}\ z\ \mathsf{then}\ x\ \mathsf{else}\ x)\ \mathit{false}$ by reducing in the right subterm, or it can take a single step to $\mathsf{if}\ z\ \mathsf{then}\ (\lambda y.\ y\ \mathit{false})\ \mathsf{else}\ (\lambda y.\ y\ \mathit{false})$ by directly β-reducing. These cannot be combined by each taking a single step, since the latter term must step once for each occurrence of x in the original function to 'catch up' to the former term.

Definition 3.3 (Weak Confluence)**.** *A reduction relation is weakly confluent if every one step split $M_1 \leftarrow M \to M_2$ can always be joined by multiple steps $M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$.*

Weak confluence is significantly easier to show, but unfortunately, it does not always imply strong confluence. Consider a term rewriting system representing flipping a coin. While the coin is active, it can be flipped from heads to tails or from tails to heads. Alternatively, at any point, it can transition to an inactive, final state of heads or tails from which it can never exit. We model this system with four terms HA, TA, H, T, and legal transitions $HA \to TA$, $TA \to HA$, $HA \to H$, $TA \to T$. This system is weakly,

but not strongly, confluent. It is weakly confluent because if the initial state M is active, in one step it can either step to the corresponding inactive state, or the opposite active state. If these do not match, they can be recombined by flipping the active state once more, and becoming inactive. On the other hand, this system is not strongly confluent because HA can step to H, and it can step to HD and then to D, but H and D are inactive and can never recombine.

We turn to the literature for a way forward in proving the confluence of the λ-calculus's β-reduction rules.

Definition 3.4 (Orthogonal Term Rewriting System)**.** *A term rewriting system is orthogonal if:*

- *The rules are all left linear*
- *There are no critical pairs*

A rule is left linear if the left hand side of each rule uses no metavariable more than once. For example, $(M = M) \to \mathit{true}$ would not be left linear, since the metavariable M appears twice. It is easy to see that the β-reduction rules are all left linear.

A critical pair of rules is a pair of rules that 'inspect' or 'match on' overlapping sets of syntax nodes. For example, if we were to add the rule $((\mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)\ M') \to (\mathsf{if}\ M\ \mathsf{then}\ N_1\ M'\ \mathsf{else}\ N_2\ M')$, it would form a critical pair with the β rule for reducing a conditional expression. The new rule matches on the conditional syntax node and the boolean value node, and the β rule matches on the application node and the conditional node. Note that the set of syntax node matched by a rule does not necessarily form a whole term, so a pair of rules such that one can apply to a subterm of a term to which the other can apply do not necessarily form a critical pair. The reduction rules we are considering are free of critical pairs. Therefore, our system is orthogonal.

Theorem 3.2. *All orthogonal term rewriting systems are strongly confluent.*

Proof. See Klop (1993), *Term Rewriting Systems.* □

Corollary 3.2.1. *β-reduction rules for λ-calculus are strongly confluent.*

Proof. As described above, the β-reduction rules are left-linear and contain no critical pairs, so the system is orthogonal and thus strongly confluent. □

4 Standardization

Definition 4.1 (Standardization)**.** *If $M \twoheadrightarrow_\beta v$ then $M \mapsto\!\!\!\!\twoheadrightarrow_\beta w \twoheadrightarrow_\beta v$.*

Definition 4.2 (Non-standard reduction)**.** *$M \rightarrowtail_\beta N$ if and only if $M \to_\beta N$ and $M \not\mapsto_\beta N$.*

We use $\rightarrowtail\!\!\!\!\twoheadrightarrow_\beta$, which is reflexive, transitive closure of $\rightarrowtail_\beta$

Lemma 4.1 (Commutation)**.** *If $M \rightarrowtail\!\!\!\!\twoheadrightarrow_\beta M_1 \twoheadrightarrow_\beta M'$ then $M \twoheadrightarrow_\beta M_2 \rightarrowtail_\beta M'$ for some M_2.*

Exercise 2: Prove commutation by proving the easier lemma: $M \rightarrowtail\!\!\!\!\twoheadrightarrow_\beta M_1 \mapsto_\beta M'$ then $M \twoheadrightarrow_\beta M_2 \rightarrowtail\!\!\!\!\twoheadrightarrow_\beta M'$.

Theorem 4.2 (Standardization)**.** *Proof.* Standardization follows from commutation. Informally, given a chain of steps $M \twoheadrightarrow_\beta v$, classify each step as either $\mapsto_\beta$ or $\rightarrowtail_\beta$. Use commutation to 'bubble up' the standard steps to the front, resulting in $M \mapsto\!\!\!\!\twoheadrightarrow_\beta N \rightarrowtail\!\!\!\!\twoheadrightarrow_\beta v$. It is easy to see by cases that nonstandard steps cannot take a non-value to a value. Therefore, N must be a value w and the proof is complete. □

5 Soundness

Theorem 5.1. *If $M =_\beta N$ then $M \approx N$.*

Proof. Assume $M =_\beta N$. Since the premise is symmetrical, we will just prove the case that $M \preccurlyeq N$. To do so we are given an arbitrary be a closing context C such that $C[M] \mapsto\!\!\!\!\twoheadrightarrow_\beta v$ for some value v. Then, we have

$$v =_\beta C[M] =_\beta C[N].$$

The first equality holds by inclusion of from $C[M] \twoheadrightarrow_\beta v$ into $=_\beta$, and the second holds by compatibility of $=_\beta$ applied to our assumption $M =_\beta N$. By confluence applied to $C[N] =_\beta v$, we have for some value w:

$$C[N] \twoheadrightarrow_\beta w \;{}_\beta\!\!\twoheadleftarrow v.$$

By standardization applied to this first transition, there must be some w' such that

$$C[N] \mapsto\!\!\!\!\twoheadrightarrow_\beta w' \twoheadrightarrow_\beta w \;{}_\beta\!\!\twoheadleftarrow v.$$

We have that $C[N] \mapsto\!\!\!\!\twoheadrightarrow_\beta w'$, so to complete the proof, we show that $w' \sim v$. This follows from the fact that they are values that reduce to the same value; no reduction rule that applies to a value changes its top-level syntax node, and therefore reduction preserves similarity. □

6 Anticipating Typed Equivalence

Returning to our examples of potential equations between programs, our definition of observational equality holds of all of these examples except that last: $\lambda x.\ not\ (not\ x) \stackrel{?}{=} \lambda x.\ x$. This cannot be proven using our current definition, as we must consider contexts in which x may be instantiated at a non-boolean term. This will result in a stuck term for the left hand side, and may not for the right hand side.

We anticipate definition a version of observational equality that is aware of the typing rules of the language, and therefore can handle this case. Here we encounter a tradeoff; the untyped relation gives less information about more terms, while the typed relation gives more information about fewer terms.