2024 OPLSS: Foundations of Programming Languages - Lecture 5

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

June 2024

1 Abstract Machines

Previous lectures use operational semantics as the main tool for explaining how a program can compute its answer. However, small-step operational semantics are terribly inefficient for practical applications, and there are two main reasons for that:

- Substitution can duplicate work an arbitrary number of times. For example, consider this beta reduction rule:

  $$(\beta \to) \quad (\lambda x.\ M)\ N \mapsto_\beta M[N/x]$$

  M can contain many occurrences of x, and N can be very large, so this substitution can explode the size of the program:

  $$((\lambda x.\ \ldots x \ldots x \ldots x)\ big) \mapsto_\beta \ldots big \ldots big \ldots big$$

- Consider the following rule:

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

  The rule does not specify how to take an arbitrary term $N = E[M]$ and locate the reducible subexpression M. Even when it is found, it may be very deep within N, and the next small step will likely require traversing N all over again. The following transversals will likely repeat previous work while navigating through N.

An *abstract machine* is an alternate form of operational semantics where the search for the next reducible term is built into the individual small-step semantics.

We define a Call-By-Name abstract machine with the following rules:

$$\begin{array}{rcl} \langle MN \parallel K \rangle & \mapsto_\beta & \langle M \parallel N \cdot K \rangle \\ \langle \lambda x.\ M \parallel N \cdot K \rangle & \mapsto_\beta & \langle M[N/x] \parallel K \rangle \end{array}$$

The form $\langle M \parallel K \rangle$ represents a machine configuration, where M is a term to reduce, and K is a continuation (represented as a stack of terms) indicating how to continue evaluation after reducing M. The configuration can be thought of as a zipper that remembers the location of the redex in the program and knows how to navigate in both directions in the term. To evaluate the application $M\ N$, we simply focus M and push N onto the continuation stack. When we get down to a lambda, we pop from the stack and make the usual substitution. Function application in this presentation still uses substitution, and therefore we still have the issue of code duplication; this could be fixed by using an environment (and closures).

2 Compilation from λ-calculus to Machine Language

There are still some issues with the abstract machine we saw in the previous section. First, the application rule is redundant; it takes one representation of an application to another. Second, the semantics makes it difficult to reason about program equivalence, since our new step relation is defined for machine states instead of programs.

We can address these issues by defining a compilation procedure from the λ-calculus to a machine language. The syntax for terms, continuations, and states is as follows:

$$\begin{aligned} \textit{Terms} \ni M, N &::= x \mid \lambda x.\ M \mid \mathsf{run}\ S \\ \textit{Continuations} \ni \quad K &::= N \cdot K \mid \mathsf{ret} \\ \textit{States} \ni \quad S &::= \langle M \parallel K \rangle \end{aligned}$$

The main point is that now, the term representation does not feature explicit application; instead, we have the run form that represents an application in compiled form using a machine state. The idea is that by translating lambda-terms into this syntax, we identify and push applications onto our continuation stack ahead of time, instead of during execution as in the section above. In addition, since reducible expressions are represented directly by machine states, we will be able to reason about program equivalence using our operational semantics.

The operational semantics of this new language is given as follows (again, defined in terms of an abstract machine with states S):

$$\begin{aligned} \langle \mathsf{run}\ S \parallel K \rangle &\mapsto_\beta S[K/\mathsf{ret}] \\ \langle \lambda x.\ M \parallel N \cdot K \rangle &\mapsto_\beta \langle M[N/x] \parallel K \rangle \end{aligned}$$

Note that we no longer have an application rule shifting a subterm into the continuation. Instead, reductions of run terms simply execute the given machine state but continue as K when the computation of that machine finishes (indicated by the ret continuation). This latter part is expressed using a substitution $S[K/\mathsf{ret}]$. Here, we must take care not to interfere with the execution of nested machine executions, so this substitution stops when it encounters another run term.

Translating lambda terms into this machine language can be done as follows:

$$\begin{aligned} [\![x]\!] &= x \\ [\![\lambda x.\ M]\!] &= \lambda x.\ [\![M]\!] \\ [\![M\ N]\!] &= \mathsf{run}\ \langle [\![M]\!] \parallel [\![N]\!] \cdot \mathsf{ret} \rangle \end{aligned}$$

Example 2.1. *Compilation of the term $[\![\lambda y.\ (\lambda x.\ x)\ y]\!]$ proceeds as follows:*

$$\begin{aligned} [\![\lambda y.\ (\lambda x.\ x)\ y]\!] &= \lambda y.\ [\![(\lambda x.\ x)\ y]\!] \\ &= \lambda y.\ \mathsf{run}\ \langle [\![\lambda x.\ x]\!] \parallel [\![y]\!] \cdot \mathsf{ret} \rangle \\ &= \lambda y.\ \mathsf{run}\ \langle (\lambda x.\ [\![x]\!]) \parallel [\![y]\!] \cdot \mathsf{ret} \rangle \\ &= \lambda y.\ \mathsf{run}\ \langle (\lambda x.\ x) \parallel [\![y]\!] \cdot \mathsf{ret} \rangle \\ &= \lambda y.\ \mathsf{run}\ \langle (\lambda x.\ x) \parallel y \cdot \mathsf{ret} \rangle \end{aligned}$$

Compilation is guaranteed to terminate, even when the term itself is nonterminating:

$$\begin{aligned} [\![\Omega]\!] &= [\![(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)]\!] \\ &= \mathsf{run}\ \langle [\![\lambda x.\ x\ x]\!] \parallel [\![\lambda x.\ x\ x]\!] \cdot \mathsf{ret} \rangle \\ &= \mathsf{run}\ \langle \lambda x.\ [\![x\ x]\!] \parallel \lambda x.\ [\![x\ x]\!] \cdot \mathsf{ret} \rangle \\ &= \mathsf{run}\ \langle \lambda x.\ \mathsf{run}\ \langle [\![x]\!] \parallel [\![x]\!] \cdot \mathsf{ret} \rangle \parallel \lambda x.\ \mathsf{run}\ \langle [\![x]\!] \parallel [\![x]\!] \cdot \mathsf{ret} \rangle \cdot \mathsf{ret} \rangle \\ &= \mathsf{run}\ \langle \lambda x.\ \mathsf{run}\ \langle x \parallel x \cdot \mathsf{ret} \rangle \parallel \lambda x.\ \mathsf{run}\ \langle x \parallel x \cdot \mathsf{ret} \rangle \cdot \mathsf{ret} \rangle \end{aligned}$$

As mentioned, we can reason about these compiled terms directly using the operational semantics. For example, for the first compilation result above, we can reason that

$$\langle (\lambda x.\ x) \parallel y \cdot \mathsf{ret} \rangle \mapsto_\beta \langle y \parallel \mathsf{ret} \rangle,$$

so that $\lambda y.\ \mathsf{run}\ \langle (\lambda x.\ x) \parallel y \cdot \mathsf{ret} \rangle$ simplifies to $\lambda y.\ \mathsf{run}\ \langle y \parallel \mathsf{ret} \rangle$. Formally, we define β-equivalence for our machine language in analogy to the definition for lambda-terms:

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

Then, we have the following key semantic property of compilation:

Theorem 2.1 (Soundness and Completeness)**.**

$$M =_\beta N \quad \textit{iff} \quad [\![M]\!] =_\beta [\![N]\!].$$

3 A Typed Interpretation of the Machine Language

The question arises of how to give types to our machine language, and what logic this corresponds to. We begin by considering machine states $S = \langle M \parallel K \rangle$. The intuition is that if M takes variables Γ to produce a type A, and if K consumes A to return with result B, then we can ascribe S the type $\Gamma \vdash \mathsf{ret} : B$.

$\boxed{S : (\Gamma \vdash \mathsf{ret} : B)}$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \mid K : A \vdash \mathsf{ret} : B}{\langle M \parallel K \rangle : (\Gamma \vdash \mathsf{ret} : B)}\ \textsc{Cut}$$

The Cut rule captures this intuition, using two separate judgments for typing terms and continuations. The former is similar to what we have already seen for the λ-calculus.

$\boxed{\Gamma \vdash M : A}$

$$\frac{}{\Gamma,\ x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma,\ x : A \vdash M : B}{\Gamma \vdash M : A \to B}\ {\to}\text{IR} \qquad \frac{S : (\Gamma \vdash \mathsf{ret} : A)}{\Gamma \vdash \mathsf{run}\ S : A}\ \textsc{Act}$$

The first two rules are the familiar ones for variables and lambda abstractions. The third uses the typing judgment for machine states in the straightforward way to type the run term.

The judgment for continuations takes a less familiar form $\Gamma \mid K : A \vdash \mathsf{ret} : B$, indicating that running the continuation K on an input A will produce an output of type B.

$\boxed{\Gamma \mid K : A \vdash \mathsf{ret} : B}$

$$\frac{}{\Gamma \mid \mathsf{ret} : A \vdash \mathsf{ret} : A}\ \textsc{CoVar} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \mid K : B \vdash \mathsf{ret} : B'}{\Gamma \mid M \cdot K : A \to B \vdash \mathsf{ret} : B'}\ {\to}\text{IL}$$

The first rule states that ret is the trivial continuation consuming A and producing A. The intuition for the second rule is that if M has type A, and K consumes B to produce B', then $M \cdot K$ expects a function $A \to B$, which it can apply to M to obtain B to pass to K, producing B'.

Note how the language has no elimination rules; instead, we have the new function typing rule $\to$IL, which introduces the function type on the left of the turnstile (as the input to be consumed by a continuation). This property is typical of a family of proof systems called *sequent calculi*. In contrast, natural deduction (which corresponds to the style of type system seen in our source lambda calculus) features only right rules, but with both introduction and elimination forms. The system at hand corresponds to a logical system called LJ created by Gerhard Gentzen. The situation can be summarized as below.

	Introduction Rules	Elimination Rules
Right Rules	SC / ND	ND
Left Rules	SC	

Table 1: Chart summarizing the forms of logical rules used in Sequent Calculi (SC) and Natural Deduction (ND). Note that other combinations of rule forms would also be possible in general.

4 First Class Control: Classical Logic

The system in the previous section featured term typings with multiple inputs (variables) and one implicit output, and continuation typings with a single (implicit) input, and a single output ret. One is tempted to bridge this asymmetry by allowing continuations with multiple outputs, and as it turns out, this has a useful interpretation both computationally and logically. We replace the run term with a new binding form $\mu\tilde{x}.\ S$, which has the same meaning, but with a *named* output $\tilde{x}$ instead of ret.

$$\begin{aligned} \textit{Terms} \ni M, N &::= x \mid \lambda x.\ M \mid \mu\tilde{x}.\ S \\ \textit{Continuations} \ni \quad K &::= N \cdot K \mid \tilde{x} \end{aligned}$$

Our typing judgments are generalized with a *covariable* environment Δ, intuitively corresponding to output channels, similarly to how Γ corresponds to inputs.

$$\boxed{S : (\Gamma \vdash \Delta)} \quad \boxed{\Gamma \vdash M : B \mid \Delta} \quad \boxed{\Gamma \mid K : A \vdash \Delta}$$

$$\frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \mid K : A \vdash \Delta}{\langle M \parallel K \rangle : (\Gamma \vdash \Delta)}\ \textsc{Cut}$$

$$\frac{}{\Gamma,\ x : A \vdash x : A \mid \Delta}\ \textsc{Var} \qquad \frac{\Gamma,\ x : A \vdash M : B \mid \Delta}{\Gamma \vdash M : A \to B \mid \Delta}\ {\to}\text{IR} \qquad \frac{S : (\Gamma \vdash \tilde{x} : A, \Delta)}{\Gamma \vdash \mu\tilde{x}.\ S : A \mid \Delta}\ \textsc{Act}$$

$$\frac{}{\Gamma \mid \tilde{x} : A \vdash \tilde{x} : A, \Delta}\ \textsc{CoVar} \qquad \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \mid K : B \vdash \Delta}{\Gamma \mid K : A \to B \vdash \Delta}\ {\to}\text{IL}$$

In our current definition, we have three types of judgments. The $S : (\Gamma \vdash \Delta)$ tells us that this machine state has access to the input channels in Γ and to the output channels in Δ. The $\Gamma \vdash M : B \mid \Delta$ asserts that the term M has type B according to our environments. The $\Gamma \mid K : A \vdash \Delta$ avers that the continuation K has type A with respect to the environments. With these additions, the system is actually capable of dealing with classical logic and corresponds to Gentzen's system LK. For example, it is possible to write non-intuitionistic terms in this system, such as a term with the type: $((A \to B) \to A) \to A$ (Peirce's law).

5 Call-By-Value and Call-By-Name Duality

We can delight in classical logic in our updated system, but in fact, a further extension is possible. We saw that in our previous system, there was a 'counterpart' of variables in the form of covariables but there weren´t duals to every construct of our language. This symmetry can be completed, and the key observation is that we can even define a symmetric evaluation relation focusing on continuations (co-terms), which will correspond to a call-by-value evaluation strategy.

$$\begin{aligned} \textit{Terms} \ni M, N &::= \ldots \mid (K, M) \\ \textit{Continuations} \ni \quad K &::= \ldots \mid \tilde{\mu}x.\ S \mid \tilde{\lambda}\tilde{x}.\ K \end{aligned}$$

The $\tilde{\mu}x.\ S$ construct binds the input channel x in S. The dual of a lambda: $\tilde{\lambda}\tilde{x}.\ K$ has the non conventional type of $A - B$. The term (K, M) is an exotic kind of pair, representing a consumer of some type B and a producer of some type A. This pair is used as input for $\tilde{\lambda}$.

The duality can be represented by an involutive negation operator defined by states, terms, and co-terms. This involutive operation turns a term into its respective co-term (and vice versa), and can be defined as:

$$\langle K \parallel M \rangle^{\perp} = \langle K^{\perp} \parallel M^{\perp} \rangle$$

$$\begin{array}{rclrcl} (\mu\tilde{x}.\ S)^{\perp} & = & (\tilde{\mu}x.\ S^{\perp}) & (\tilde{\mu}x.\ S)^{\perp} & = & \mu\tilde{x}.\ S^{\perp} \\ (\lambda x.\ M)^{\perp} & = & (\tilde{\lambda}\tilde{x}.\ M^{\perp}) & (M \cdot K)^{\perp} & = & (M^{\perp}, K^{\perp}) \\ (K, M)^{\perp} & = & (K^{\perp} \cdot M^{\perp}) & (\tilde{\lambda}\tilde{x}.\ K)^{\perp} & = & (\lambda x.\ K^{\perp}) \end{array}$$

The interesting fact is that if we negate a call-by-name machine state (in other words, if we invert the positions of term and co-terms), we obtain a call-by-value machine state. Both states represent the same computation but proceed with different evaluation strategies.

Theorem 5.1 (Operational Duality)**.** *$S \mapsto_{\beta} S'$ under the call-by-value semantics if and only if $S^{\perp} \mapsto_{\beta} S'^{\perp}$ under the call-by-name semantics.*