Formal Foundations of Programming Languages
Lecture Notes

Ralf Jung Max Vistrup

ETH Zurich

based on

Semantics of Type Systems

Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung,
Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey

Contents

1 Simply Typed Lambda Calculus **2**
1.1 Operational Semantics 3
1.2 The Untyped λ-calculus 5

1 Simply Typed Lambda Calculus

$$\begin{array}{rrcl} \text{Variables} & x, y & & \ldots \\ \text{Runtime Terms} & & e & ::= x \mid \lambda x.\, e \mid e_1\, e_2 \mid e_1 + e_2 \mid \overline{n} \\ \text{(Runtime) Values} & & v & ::= \lambda x.\, e \mid \overline{n} \end{array}$$

Variables and Substitution In the simply typed λ-calculus, variables and substitution are instrumental to define how terms are evaluated (called the *operational semantics*). During evaluation, if we apply a λ-abstraction $\lambda x.\, e$ to a value v, then the variable x is *substituted* with the value v in e. For example, the term $(\lambda x.\, x + \overline{1})\, \overline{41}$ proceeds with $\overline{41} + \overline{1}$ in the next step of the evaluation. This step is commonly called "β-reduction" ("beta-reduction").

We write $e[e'/x]$ for the substitution operation that replaces x with e' in e, and we pronounce this "e with e' for x". We define it recursively by:

$$\begin{array}{rll} y[e'/x] := e' & & \text{if } x = y \\ y[e'/x] := y & & \text{if } x \neq y \\ (\lambda y.\, e)[e'/x] := \lambda y.\, e & & \text{if } x = y \\ (\lambda y.\, e)[e'/x] := \lambda y.\, (e[e'/x]) & & \text{if } x \neq y \\ (e_1\, e_2)[e'/x] := (e_1[e'/x])\, (e_2[e'/x]) & & \\ (e_1 + e_2)[e'/x] := (e_1[e'/x]) + (e_2[e'/x]) & & \\ \overline{n}[e'/x] := \overline{n} & & \end{array}$$

Getting substitution right can be tricky. This definition is typically considered incorrect. To explain what goes wrong, we have to distinguish between *free* and *bound* variables: A variable x is bound in a term if it appears inside of a binder "λx" (*e.g.*, x is bound in $\lambda x.\, x + y$). All other variables are called *free* (*e.g.*, y is free in $\lambda x.\, x + y$). Formally, the set $\mathsf{fv}(e)$ of free variables in e is defined as

$$\begin{aligned} \mathsf{fv}(x) &:= \{x\} \\ \mathsf{fv}(\lambda x.\, e) &:= \mathsf{fv}(e) \setminus \{x\} \\ \mathsf{fv}(e_1\, e_2) &:= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(e_1 + e_2) &:= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(\overline{n}) &:= \emptyset \end{aligned}$$

The problem with the naive definition of substitution above is called *variable capturing*. Variable capturing occurs if we insert an expression with a free variable such as $\overline{2} + x$ into an expression which binds the free variable. For example, if we naively substitute $\overline{2} + x$ for y in $\lambda x.\, y + x$, then we obtain $\lambda x.\, (\overline{2} + x) + x$. Since x was free in $\overline{2} + x$ but is bound in $\lambda x.\, (\overline{2} + x) + x$, one speaks of variable capturing. Variable capturing is problematic, because the programmer of $\lambda y, x.\, y + x$ would have to anticipate which variables are free in the function argument inserted for y.

To avoid variable capturing, a correct substitution renames bound variables where conflicts arise. For example, $(\lambda x.\, y{+}x)[\overline{2}{+}x/y]$ should result in $\lambda z.\, (\overline{2}{+}x){+}z$, such that x remains free. Unfortunately, defining (and reasoning about) a substitution operation that properly renames bound variables is oftentimes tedious, especially in proof assistants. Thus, on paper, it is standard to assume and follow *Barendregt's variable convention* [3]: all bound and free variables are distinct and this invariant is maintained implicitly.

Since we *mechanize* our proofs in Coq, we cannot assume Barendregt's variable convention. Instead, we use the (slightly broken) substitution operation above. In our use cases, the substitution will only insert *closed* terms (*i.e.*, terms without any free variables: $\mathsf{fv}(e) = \emptyset$), which avoids the problem of variable capture entirely. (In later sections, Section **??**, we will discuss the more complicated DeBruijn representation of terms with binders, which simplifies defining a correct, capture-avoiding substitution.)

1.1 Operational Semantics

In order to reason about programs, we have to assign a semantics to them. In the following, we assign an *operational semantics* to programs, meaning we describe how runtime terms are evaluated. We distinguish three different operational semantics: *big-step semantics*, *structural semantics*, and *contextual semantics*. They are all equivalent but distinct ways of expressing how our terms compute.

Big-Step Semantics

$\boxed{e \downarrow v}$

The big-step semantics are arguably the "most natural" semantics: they directly define when a term e computes to a value v.

$$\textsc{literal}\ \frac{}{\overline{n} \downarrow \overline{n}} \qquad \textsc{lambda}\ \frac{}{\lambda x.\, e \downarrow \lambda x.\, e} \qquad \textsc{app}\ \frac{e_1 \downarrow \lambda x.\, e \qquad e_2 \downarrow v_2 \qquad e[v_2/x] \downarrow v}{e_1\, e_2 \downarrow v} \qquad \textsc{plus}\ \frac{e_1 \downarrow \overline{n_1} \qquad e_2 \downarrow \overline{n_2}}{e_1 + e_2 \downarrow \overline{n_1 + n_2}}$$

Exercise 1 Prove that the big-step semantics $e \downarrow v$ is *deterministic*. That is, show that if $e \downarrow v$ and $e \downarrow v''$, then $v = v'$. •

Small-step semantics

Whereas a big-step semantics takes a single "big step" directly from a term to its final value, a small-step semantics defines a step-by-step process whereby the term is slowly reduced further and further, until it eventually reaches a value. Small-step semantics look more complicated at first, but they are needed when considering infinite executions, or to model concurrency (where the individual steps of multiple computations can interleave).

Small-step semantics come in two flavors, "structural" and "contextual". We will discuss structural semantics (often called SOS, "Structured Operational Semantics") first.

Structural Semantics

$\boxed{e \succ e'}$

We define a structural *call-by-value, weak, right-to-left* operational semantics on our runtime terms:

$$\textsc{app-struct-r}\ \frac{e_2 \succ e_2'}{e_1\, e_2 \succ e_1\, e_2'} \qquad \textsc{app-struct-l}\ \frac{e_1 \succ e_1'}{e_1\, v_2 \succ e_1'\, v_2} \qquad \textsc{beta}\ (\lambda x.\, e)\, v \succ e[v/x]$$

$$\textsc{plus-struct-l}\ \frac{e_1 \succ e_1'}{e_1 + v_2 \succ e_1' + v_2} \qquad \textsc{plus-struct-r}\ \frac{e_2 \succ e_2'}{e_1 + e_2 \succ e_1 + e_2'} \qquad \textsc{plus}\ \overline{n} + \overline{m} \succ \overline{n + m}$$

The three adjectives have the following meaning:

- *Right-to-left:* in a term like $e_1 + e_2$, we evaluate e_2 to a value before we even begin evaluating e_1. This is reflected in PLUS-STRUCT-L by requiring the right-hand operand to be a value for the rule to apply.
- *Call-by-value:* we evaluate the argument of a function to a value before doing substitution. This is reflected in BETA requiring the argument to be a value, and in APP-STRUCT-R letting us evaluate the argument *before* doing β-reduction.
- *Weak:* we do *not* allow reduction below λ abstractions. This is reflected by *not* having a structural rule for $\lambda x.\, e$. In contrast, a *strong* semantics would let us reduce $\lambda x.\, (\lambda y.\, y)\, x$ to $\lambda x.\, x$. However, given that our substitution operation is not capture-avoiding, we cannot allow strong reduction in our calculus.

Exercise 2 Prove that the structural semantics $e \succ e'$ is *deterministic*. That is, show that if $e \succ e'$ and $e \succ e''$, then $e' = e''$. •

Exercise 3 The semantics $e \succ e'$ is a *call-by-value* semantics, meaning arguments are evaluated to values before they are inserted into lambda abstractions. The call-by-value approach is used by many programming languages (e.g., Java, C, Standard ML, and OCaml). Alternatively, one can defer the evaluation of function arguments and, instead, insert their unevaluated form directly into lambda abstractions. This style of operational semantics is called *call-by-name* semantics and is followed by some functional languages (e.g., Haskell). The core rule of this semantics is:

$$\frac{}{(\lambda x.\, e)\, e' \succ_{\mathsf{cbn}} e[e'/x]}\ \textsc{cbn-beta}$$

1. Complete the definition of $e \succ_{\mathsf{cbn}} e'$ and give one expression, which evaluates to different values under *call-by-name* and *call-by-value* semantics.
   Hint: For call-by-name, the left side of an application has to be evaluated first.
2. Prove that $e \succ_{\mathsf{cbn}} e'$ is deterministic.

•

Exercise 4 Prove that the big-step and the small-step semantics $e \succ e'$ are equivalent:

$$e \downarrow v \quad \textit{iff} \quad e \succ^* v$$

Here, $\succ^*$ is the reflexive transitive closure of $\succ$, *i.e.*, $e \succ^* v$ means that we can step from e to v using a sequence of reduction steps (including possibly using no steps at all). •

Exercise 5 The evaluation order for $e \succ e'$ is right-to-left. We are now going to consider left-to-right evaluation order.

1. Define a semantics $e \succ_{\mathsf{ltr}} e'$, which evaluates expressions in left-to-right order.
2. Pick terms e and e' such that $e \succ_{\mathsf{ltr}} e'$ but not $e \succ e'$.
3. Show that both are equivalent if we evaluate to values.
   Hint: It suffices to show $e \succ^*_{\mathsf{ltr}} v \quad \textit{iff} \quad e \downarrow v$.
4. Think of the programming languages you have encountered in your past. Is it in one of them possible to obtain different results, depending on left-to-right or right-to-left evaluation order?

•

Contextual Semantics

The structural semantics $e \succ e'$ has two kinds of rules: (1) rules such as APP-STRUCT-L, which descend into the term to find the next subterm to reduce (which is called a *redex*) and (2) rules such as BETA and PLUS, which reduce redexes. Next, we define a third operational semantics, the contextual operational semantics $e_1 \rightsquigarrow e_2$, which separates the search for the redex (*i.e.*, structurally descending in the term) from the reduction. While separating redex search and reduction does not have any immediate benefits for us at the moment, it will lead to more elegant reasoning principles later on.

For the contextual semantics, we first define evaluation contexts:

$$\text{Evaluation Contexts} \quad K \ ::= \ \bullet \ \mid \ K\, v \ \mid \ e\, K \ \mid \ K + v \ \mid \ e + K$$

Evaluation contexts are expressions with a hole (*e.g.*, $\bullet + \overline{41}$), which describe where in the expression the next redex can be found. We can fill the hole with an expression using the following function:

Context Filling $\boxed{K[e]}$

$$\begin{aligned} \bullet[e] &:= e & (K + v)[e] &:= K[e] + v \\ (K\, v)[e] &:= (K[e])\, v & (e' + K)[e] &:= e' + K[e] \\ (e'\, K)[e] &:= e'\, (K[e]) \end{aligned}$$

Base reduction and contextual reduction $\boxed{e_1 \rightsquigarrow_{\mathrm{b}} e_2 \text{ and } e_1 \rightsquigarrow e_2}$

$$\textsc{beta}\ \ (\lambda x.\, e)\, v \rightsquigarrow_{\mathrm{b}} e[v/x] \qquad \textsc{plus}\ \ \overline{n} + \overline{m} \rightsquigarrow_{\mathrm{b}} \overline{n+m} \qquad \textsc{ctx}\ \ \frac{e_1 \rightsquigarrow_{\mathrm{b}} e_2}{K[e_1] \rightsquigarrow K[e_2]}$$

Lemma 1 (Context Lifting)**.** *If $e_1 \rightsquigarrow e_2$, then $K[e_1] \rightsquigarrow K[e_2]$.*

Proof. Assume that $e_1 \rightsquigarrow e_2$. By inversion, there exist K', e_1', e_2' such that $e_1' \rightsquigarrow_{\mathrm{b}} e_2'$ and $e_1 = K'[e_1']$ and $e_2 = K'[e_2']$. To construct a proof of $K[K'[e_1']] \rightsquigarrow K[K'[e_2']]$, we essentially need to compose the two contexts in order to apply CTX.

We define a context composition operation in the following. We can then close this proof by Lemma 2. □

Context Composition $\boxed{K_1 \circ K_2}$

$$\begin{aligned} \bullet \circ K_2 &:= K_2 & (K + v) \circ K_2 &:= (K \circ K_2) + v \\ (K\, v) \circ K_2 &:= (K \circ K_2)\, v & (e' + K) \circ K_2 &:= e' + (K \circ K_2) \\ (e'\, K) \circ K_2 &:= e'\, (K \circ K_2) \end{aligned}$$

Lemma 2. $K_1[K_2[e]] = (K_1 \circ K_2)[e]$

Proof. By induction on K_1. □

Exercise 6 As an example for a reasoning principle that can be stated more elegantly in the contextual semantics than the structural semantics, we consider context lifting for the reflexive transitive closure of $\rightsquigarrow$. Prove the following: if $e_1 \rightsquigarrow^* e_2$, then $K[e_1] \rightsquigarrow^* K[e_2]$.

To obtain the same result in the structural semantics, we'd have to state four lemmas: one for each structural rule. Having a notion of "evaluation contexts" gives us some extra vocabulary that makes these kinds of properties a lot easier to state. •

Exercise 7 Prove that the structural and the contextual semantics are equivalent:

$$e \succ e' \quad \textit{iff} \quad e \rightsquigarrow e'$$

•

1.2 The Untyped λ-calculus

Before we start to integrate *types* into our calculus (in Section **??**), we first examine the *untyped* version of the lambda calculus. The untyped lambda calculus, even without addition and natural numbers, is quite expressive computationally—it is Turing complete! We will now explore its computational power in the fragment $e ::= x \mid \lambda x.\, e \mid e_1\, e_2$. The slogan is: *All you need are lambdas, variables, and beta reduction.*

To get started, let us write a term that causes an infinite reduction.

$$\omega := \lambda x.\, xx \qquad\qquad \Omega := \omega\omega$$

ω applies its argument to itself, and Ω applies ω to itself. Let us try out what happens when we evaluate Ω.

$$\Omega = \omega\omega = (\lambda x.\, xx)\omega \succ (xx)[\omega/x] = \omega\omega = \Omega$$

As it turns out, Ω reduces to itself! Thus, there are terms in the untyped lambda calculus such as Ω which *diverge* (*i.e.*, their reduction chains do not terminate). Note that we had to use the small-step operational semantics for this; big-step semantics can only talk about terminating executions.

Scott encodings We can not only write diverging terms, but we can also encode inductive data types in the untyped lambda calculus. For example, we can encode natural numbers in their Peano representation (*i.e.*, with the constructors 0 and S) as lambda terms. The basic idea is to interpret natural numbers as "case distinctions". That is, each number will be an abstraction with two arguments, z and s, the "zero" and "successor" cases. If the number is 0, then it will return the argument z. If the number is $\mathsf{S}\,n$, then it will return $s\,n$, *i.e.*, it will call x applied to the predecessor n. You should think of $n\,z\,s$ as being equivalent to the following Coq code:

```
match n with
  O => z
| S n => s n
end
```

Now we can define our first numbers. We will write $\widetilde{0}$ for the Scott encoding of 0, and so on:

$$\widetilde{0} := \lambda z, s.\, z \qquad\qquad \widetilde{1} := \lambda z, s.\, s\,\widetilde{0} \qquad\qquad \widetilde{2} := \lambda z, s.\, s\,\widetilde{1}$$

(We use $\lambda z, s.\, z$ as a short-hand for $\lambda z.\, \lambda s.\, z$.)

Following this principle, we can define the encoding of an arbitrary number as a lambda term:

$$\widetilde{0} = \widetilde{0}$$
$$\widetilde{\mathsf{S}\,n} = \lambda z, s.\, s\ \widetilde{n}$$

Note that $\widetilde{n}$ is a value of each n, *i.e.*, it does not reduce any further as a lambda term.

We can also define the successor operation as a function in the calculus:

$$\widetilde{S} := \lambda n.\, \lambda z, s.\, s\,n$$

Convince yourself that $\widetilde{S}\,\widetilde{1}$ reduces to $\widetilde{2}$, and in general $\widetilde{S}\,\widetilde{n}$ reduces to $\widetilde{S\,n}$.

We can use this representation, known as the Scott encoding, to compute with natural numbers. That is, since we can do a case analysis on the numbers *by definition*, we can distinguish them and compute different results depending on the case. For example, the function $\mathsf{pred} = \lambda n.\, n\ \widetilde{0}\ (\lambda n'.\, n')$ computes the predecessor of a Scott encoded natural number. (Note that function application is left-associative, so $n\ \widetilde{0}\ e$ is short for $(n\ \widetilde{0})\ e$.)

Exercise 8 Define a function on Scott encoded natural numbers which returns the identity for 0 and diverges for every other number. •

If we want to define slightly more interesting functions on our numbers such as addition $\mathsf{add}\ n\ m$, we run into a problem. The naive definition of multiplication by 2 would be:

$$\mathsf{mul2} := \lambda n.\ n\ \widetilde{0}\ (\lambda n'.\ \widetilde{S}(\widetilde{S}(\mathsf{mul2}\ n')))$$

Remember that to read (and write) the definitions, it is helpful to turn them into Coq `match` statements as explained above.

Unfortunately, this definition is broken! The term that we want to define, mul2, occurs in its own definition on the right hand side. To fix this problem, we are now going to develop a mechanism to add recursion to the untyped lambda calculus.

Recursion To enable recursion, we define a recursion operator fix (sometimes called fixpoint combinator or Y-combinator). The idea of fix is that given a template $F := \lambda rec, x.\ e$ of the recursive function that we want to define, where *rec* is a placeholder for recursive calls, the expression $\mathsf{fix}\ F\ v$ reduces to $F\ (\mathsf{fix}\ F)\ v$. In other words, we end up calling F such that *rec* becomes $\mathsf{fix}\ F$ (and the value v is just forwarded).

In other words, the desired reduction behavior of fix is

$$\mathsf{fix}\ F\ v \succ^* F\ (\mathsf{fix}\ F)\ v$$

We will define fix below. Beforehand, let us explore how we can define recursive functions such as mul2 with fix. We first define the template MUL2, and then we take the fixpoint to obtain mul2:

$$\begin{aligned}\mathsf{MUL2} &:= \lambda rec, \lambda n.\ n\ \widetilde{0}\ (\lambda n'.\ \widetilde{S}(\widetilde{S}(rec\ n')))\\ \mathsf{mul2} &:= \mathsf{fix}\ \mathsf{MUL2}\end{aligned}$$

Why does this make sense? Let us consider what happens when reducing $\mathsf{mul2}\ \widetilde{1}$:

$$\begin{array}{lcll}
\mathsf{mul2}\ \widetilde{1} & = & \mathsf{fix}\ \mathsf{MUL2}\ \widetilde{1} & \text{by the definition of mul2}\\
& \succ^* & \mathsf{MUL2}\ (\mathsf{fix}\ \mathsf{MUL2})\ \widetilde{1} & \text{by the fix reduction rule}\\
& = & \mathsf{MUL2}\ \mathsf{mul2}\ \widetilde{1} & \text{by the definition of mul2}\\
& \succ^* & \widetilde{1}\ \widetilde{0}\ (\lambda n'.\ \widetilde{S}(\widetilde{S}(\mathsf{mul2}\ n'))) & \text{by the definition of MUL2 and two } \beta\text{-reductions}\\
& \succ^* & (\lambda n'.\ \widetilde{S}(\widetilde{S}(\mathsf{mul2}\ n')))\ \widetilde{0} & \text{by the definition of } \widetilde{1} \text{ and two } \beta\text{-reductions}\\
& \succ & \widetilde{S}(\widetilde{S}(\mathsf{mul2}\ \widetilde{0})) & \text{by another } \beta\text{-reduction}\\
& \succ^* & \widetilde{S}(\widetilde{S}(\mathsf{MUL2}\ \mathsf{mul2}\ \widetilde{0})) & \text{by the fix rule and the definition of mul2}\\
& \succ^* & \widetilde{S}(\widetilde{S}(\widetilde{0}\ \widetilde{0}\ (\lambda n'.\ \widetilde{S}(\widetilde{S}(\mathsf{mul2}\ n'))))) & \text{by the definition of MUL2 and two } \beta\text{-reductions}\\
& \succ^* & \widetilde{S}(\widetilde{S}(\widetilde{0})) & \text{by the definition of } \widetilde{0} \text{ and two } \beta\text{-reductions}\\
& \succ^* & \widetilde{2} & \text{by the definition of } \widetilde{S} \text{ and two } \beta\text{-reductions}
\end{array}$$

In fact, we can prove that mul2 has the desired computational behavior:

Lemma 3. $\mathsf{mul2}\ \widetilde{n} \succ^* \widetilde{2 * n}$

To define fix, we will abstract over the template and assume it is some abstract value F. The definition of $\mathsf{fix}\ F$ (where fix is parametric in F) consists of two parts:

$$\begin{aligned}\mathsf{fix}\ F &:= \lambda x.\ \mathsf{fix}'\ \mathsf{fix}'\ F\ x\\ \mathsf{fix}' &:= \lambda f, F.\ F\ (\lambda x.\ f\ f\ F\ x)\end{aligned}$$

The idea is that fix is defined using a term fix', which relies on self-application (like Ω) to implement recursion. More specifically, we provide fix' with itself, the template F, and the function argument x.

In the definition of fix', we are given fix' as f and the template F. The argument for x is omitted to get the right reduction behavior (as we will see below). Given these arguments, we want to apply the template F to $\mathsf{fix}\ F$. What this means in the scope of fix' is that we apply F to the term $\lambda x.\ f\ f\ F\ x$.

With this definition, $\mathsf{fix}\ F$ has the right reduction behavior:

$$\begin{aligned}
\mathsf{fix}\ F\ v &= (\lambda x.\ \mathsf{fix}'\ \mathsf{fix}'\ F\ x)\ v \\
&\succ \mathsf{fix}'\ \mathsf{fix}'\ F\ v \\
&\succ (\lambda F.\ F\ (\lambda x.\ \mathsf{fix}'\ \mathsf{fix}'\ F\ x))\ F\ v \\
&\succ F\ (\lambda x.\ \mathsf{fix}'\ \mathsf{fix}'\ F\ x)\ v \\
&= F\ (\mathsf{fix}\ F)\ v
\end{aligned}$$

With first-order inductive data types and recursion, one can define basic arithmetic operations (*e.g.*, addition, multiplication) and build up larger programs. In fact, we have now seen all the basic building blocks that are needed to prove that the untyped lambda calculus is Turing complete (which we will not do in this course).

References

[1] A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 2001.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd revised edition, 1984.

[4] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP*, 1983.

[5] S. Schäfer, T. Tebbi, and G. Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *Interactive Theorem Proving*, 2015.

[6] P. Wadler. Theorems for free! In *FPCA*, 1989.