Chapter 18

Variables and Variable Binding

In this set of notes, we learn about *variables* and *variable bindings* in programming languages. Along the way, we learn a bunch more about the nature of recursive function definitions.

So far the programs that we have been writing essentially boil down to basic arithmetic. We can write down values and computations on those values, but we have no forms of *abstraction* at all. For example, you may find yourself writing the same expression over and over again, and realize that it's always going to be the same. It would be nice if we only had to write that expression down once and then had a way to reuse it. This is exactly what *variables* (a.k.a. *identifiers*) are for! They let us rewrite an expression like

```
(if (zero? (- (+ 3 2) 1))
    (+ 3 2)
    (- (+ 3 2) 1))
```

in a form that makes it explicit that we're repeating ourselves. Rather than write the expression `(+ 3 2)` repeatedly, we say once that the variable `x` stands for the expression, and refer to that variable wherever we mean to repeat the same notion:

```
(let ([x (+ 3 2)])
  (if (zero? (- x 1))
      x
      (- x 1)))
```

Here we *bind* the value of `(+ 3 2)` to the variable `x` and refer to that variable wherever we want the value.

So we can use name binding to avoid unnecessary duplication in two senses. In the first sense, we can abbreviate our program and say things once and only once. In the second sense, some language implementations can take advantage of this form of expression abstraction to improve runtime performance. Depending on the particular language design (as we discuss later), the implementation may be able to[1] evaluate the expression only once, cache the result somewhere, and then look up the cached value whenever it is needed. This can substantially improve the performance of programs (but not always!).

Notice that I am distinguishing between the *semantics* of this language feature (i.e., the conceptual meaning of variable binding as abstracting expressions), and the *pragmatics* of the feature (i.e., how this conception of variable binding can facilitate a nice implementation strategy). In practice, language design choices can be driven in either direction. Sometimes the desired semantics for a language feature suggest a particular implementation strategy. Other times the current state of the art of implementation (or experience with implementing prior languages)—or intrinsic limitations imposed by computability or complexity theory—may drive decisions about the semantics of language features. Here we focus on the (often less appreciated) direction from semantics to pragmatics.

[1] or *have* to!

18.1 Naïve Substitution

Before we add name binding to the language, let's consider what it means to have variable references in the body of such a binding. Consider our last example above. Intuitively, we want to use the meaning of `(+ 3 2)` everywhere that the expression refers to `x`. Taking `5` as the meaning of `(+ 3 2)`,[2] we want to *substitute* `5` for `x` in the body of the `let` expression.

Let's formalize *just this concept* of substitution in the context of our language of Boolean expressions.

$$\begin{array}{lll} t \in \textsc{Term}, & x \in \textsc{Var} & \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & \boxed{x} \end{array}$$

The set $\textsc{Var}$ is some infinite set of identifiers, where the only thing we care about is that we can tell one variable apart from another. The $\textsc{Term}$ x is a *variable reference*, which indicates where an *argument* will be eventually substituted. A useful way to think about variable references is that, rather than treating variables as $\textsc{Term}$s, to think of variables as *indices* on terms. So whenever you see the term x, we *really* see it as standing for the $\textsc{Tree}$ $\mathsf{var}[x]()$, where var is an $\textsc{Atom}$ but it is paired with a variable $x \in \textsc{Var}$.[3] This distinction may seem a bit strange now, and it is indeed not technically necessary, but when we get to describing variable binding using the let form (a.k.a. let-binding), it will hopefully make a bit more sense why we are setting things up this way.

Now here, it's worth taking an aside to talk about *metavariables* and *object variables.* Throughout the course, we have talked about metavariables like t and n, which stand for $\textsc{Term}$s and numbers in the object language respectively. Now we have our first object language where the language itself has a notion of "variables." We have to be clear that these are not the same things as metavariables, and the implications are important. When talking about specific object language variables, we'll use blue sans-serif font variable names like x and y. We don't *really* care about the syntactic nuances of a particular programming language, like whether you can have a numeral as the first character of a variable name, or whether you can have Unicode characters in your variable names, etc. But we need SOME way of writing examples, hence our blue object variables. Metavariables like x_1 and x_2 refer to metalanguage variables that represent object language variables like x and y. Probably the most important thing to keep in mind, which leads to no end of mistakes, is that in a given context, an individual metavariable always refers to the same thing. So $x = x$ is going to always be true. That's the funny thing about variables, they don't really vary (at least not the way those thingies in C that we, perhaps more justifably, call variables do). On the other hand, given two different metavariables, x_1 and x_2, we don't know a priori whether they refer to the same metavariable or not. If we want them to definitely be different, we must *explicitly* say $x_1 \neq x_2$. Sometimes texts leave that side condition out under the assumption that it's obvious from the context.[4] In those circumstances it's really important to make sure that you notice that this may be what's going on, and if so to make a note of it to yourself as you try to understand what you are reading. Below, we'll see this come up.

We model substitution using functions from $\textsc{Term}$s to $\textsc{Term}$s. The strategy that we take, though, is to define a different substitution function for *each* instance of substitution. The idea is that once you've seen one version of this function, you can see exactly how you would define any other (so you then know that each such function exists, is unique, and satisfies the equations used to describe it). We take this approach because it lets us use the tools that we currently have available to prove that these functions exist.

For instance, let's define a function $\underline{[\mathsf{x} \Mapsto \mathsf{0}]} : \textsc{Term} \to \textsc{Term}$ that substitutes $\mathsf{0}$ for every instance of x in a term. Note that $\underline{[\mathsf{x} \Mapsto 0]}$ is just an evocative *name* for this functon, just like *eval* or *dom.* The $\mathsf{0}$ and x don't actually *do* anything here. We could replace the name with F and it would still be the same function. For instance, we expect the following equation to hold:

$$\underline{[\mathsf{x} \Mapsto \mathsf{0}]}(\mathsf{if\ false\ then\ x\ else}\ (\mathsf{x} + +\mathsf{1})) = \mathsf{if\ false\ then\ 0\ else}\ (\mathsf{0} + +\mathsf{1}).$$

We define this function recursively, first in long-hand style. Here is the principle of definition by recursion for the language augmented with variables:

---

[2] Stay tuned! We revisit this assumption below when we discuss *by-name* evaluation.

[3] In this context, When we refer to $\textsc{Tree}$, we are *really* referring to $\textsc{Tree}[\textsc{Atom} \cup (\{\, \mathsf{var} \,\} \times \textsc{Var})]$, representing the $\textsc{Var}$-indexed var as a pair.

[4] I've been guilty of doing this, I'll admit it! Sorry...

Proposition 68 (Principle of Definition by Recursion on terms $t \in \textsc{Term}$)**.** *Let S be a set and $s_t, s_f \in S$ be two elements,*

$$H_{\mathit{if}} : S \times S \times S \to S$$

be a function on S, and

$$\boxed{H_{var} : \textsc{Var} \to S.}$$

Then there exists a unique function

$$F : \textsc{Term} \to S$$

such that

1. $F(\mathit{true}) = s_t$*;*
2. $F(\mathit{false}) = s_f$*;*
3. $F(\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3) = H_{\mathit{if}}(F(t_1), F(t_2), F(t_3))$*;*
4. $\boxed{F(x) = H_{var}(x)}$.

The new components of the principle are marked in grey above. The H_{var} function on variables $\textsc{Var}$ is responsible for handling *all* possible variable references. Essentially this function is analogous to the s_t and s_f constants, which handle the two individual constants. Recall that earlier I mentioned that we treat variable references as $\textsc{Var}$-indexed trees $\mathsf{var}(x)$. Here is where we see how this point of view is played out. Mind you this approach is just a way of thinking about the idea of variables all being the same kind of term. We could do the same thing with numbers n if the language had them.

So now let's define $\underline{[\mathsf{x} \mapsto \mathsf{0}]}$. We define the function by applying the principle of definition by recursion to the following components:

1. $S = \textsc{Term}$;
2. $s_t = \mathsf{true}$;
3. $s_f = \mathsf{false}$;
4. $H_{\mathit{if}}(t_1, t_2, t_3) = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$;
5. $H_{var}(\mathsf{x}) = \mathsf{0}$
   $H_{var}(x_0) = x_0 \quad \text{if } x_0 \neq \mathsf{x}$.

Be careful not to be confused by the fact that we chose $S = \textsc{Term}$, which is necessary to define a function from $\textsc{Terms}$ to $\textsc{Terms}$.

Substituting these components into the principle itself (and breaking the cases into separate equations as is standard practice), we get the following shorthand definition:[5]

$$\begin{aligned}
\underline{[\mathsf{x} \mapsto \mathsf{0}]} &: \textsc{Term} \to \textsc{Term} \\
\underline{[\mathsf{x} \mapsto \mathsf{0}]}\mathsf{x} &= \mathsf{0} \\
\underline{[\mathsf{x} \mapsto \mathsf{0}]}x_0 &= x_0 \qquad \text{if } x_0 \neq \mathsf{x} \\
\underline{[\mathsf{x} \mapsto \mathsf{0}]}\mathsf{true} &= \mathsf{true} \\
\underline{[\mathsf{x} \mapsto \mathsf{0}]}\mathsf{false} &= \mathsf{false} \\
\underline{[\mathsf{x} \mapsto \mathsf{0}]}(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= \mathsf{if}\ (\underline{[\mathsf{x} \mapsto \mathsf{0}]}t_1)\ \mathsf{then}\ (\underline{[\mathsf{x} \mapsto \mathsf{0}]}t_2)\ \mathsf{else}\ (\underline{[\mathsf{x} \mapsto \mathsf{0}]}t_3).
\end{aligned}$$

If we extend our language to include arithmetic expressions, and follow the same recipe, then we can show that the example above exactly fits this model.

[5]It is a common notational style to not wrap the substitution argument in parentheses.

Now to generalize: one thing that we can immediately see is that given any TERM t and any VAR x, we can define a function $\underline{[x \mapsto t]} : \text{TERM} \to \text{TERM}$ by choosing a different $H_{var}(x)$ function for the principle of recursion. This fact justifies a corresponding generic function definition.

$$[\cdot \mapsto \cdot]\cdot : \text{VAR} \times \text{TERM} \to \text{TERM} \to \text{TERM}$$
$$[x \mapsto t]t_0 = \underline{[x \mapsto t]}t_0$$

For this definition, we're using slightly different notation (no underlines!) just to make it clear that we are defining a separate general substitution function using the hard-wired individual functions. We call this function *naïve* substitution as a hint that we are working toward a fully satisfactory notion of *capture-avoiding* substitution, which comes later. Now our approach to defining naïve substitution here may seem contrived: defining individual substitution functions first and then defining the big mother of functions later. Well to be honest, *it is*! It's possible to define naïve substitution outright as a group of recursive equations without a side step through individual substitution functions, and as you'll see the definition looks pretty familiar. However, we have not yet learned the tools that we would need to claim the propriety of this equational definition without also providing a proof that the equations indeed define a function (i.e., a proof along the lines of the proof of the principle of definition by recursion). Later in the course we'll learn more tools so that we can justify such a definition without writing down a separate proof. However, we know enough to be able to define substitution this way, and externally we could prove some equational theorems about it that make it easier to use directly.

Proposition 69. *Naïve substitution satisfies the following equations.*

$$\begin{aligned}
[x \mapsto t]x &= t \\
[x \mapsto t]x_0 &= x_0 \qquad \textit{if } x_0 \neq x \\
[x \mapsto t]\mathit{true} &= \mathit{true} \\
[x \mapsto t]\mathit{false} &= \mathit{false} \\
[x \mapsto t](\mathit{if}\, t_1 \;\mathit{then}\; t_2 \;\mathit{else}\; t_3) &= \mathit{if}\;([x \mapsto t]t_1)\;\mathit{then}\;([x \mapsto t]t_2)\;\mathit{else}\;([x \mapsto t]t_3).
\end{aligned}$$

Funny, this *proposition* look almost exactly like our *definition* of the $\underline{[\mathsf{x} \mapsto \mathsf{0}]}$ function. Furthermore, we can prove that there is *exactly one* function in $\text{VAR} \times \text{TERM} \to \text{TERM} \to \text{TERM}$ that satisfies these equations, so this could in fact be our definition, but we have to prove that this is true before we can assert that this is a definition. Remember: the principle of definition by recursion once-and-for-all proves this uniqueness property for a certain class of function definitions. Unfortunately, this set of equations cannot be recast in a way that fits the *schema* put forth by that definition (you should ask yourself and answer: what goes wrong?). So for now we settle for our indirect way of defining the function and separately proving that the above proposition holds. Later, though, we'll be able to easily justify these equations as our definition of substitution.

18.2 Let Bindings

Now that we have a mathematical model for what it means to substitute a language term for a variable reference, our next step is to add let bindings to our language!

First, let's introduce our notation for let bindings.

$$t ::= \ldots \mid \mathsf{let}\; x = t \;\mathsf{in}\; t$$

The idea behind this expression is that $\mathsf{let}\; x = t_1 \;\mathsf{in}\; t_2$ means that within the expression t_2 we are to take the variable x to stand for the result of the expression t_1. Referring back to tree notation, we can think of this as $\mathsf{let}[x](t, t)$, where again the variable x is just a VAR, not a TERM. Another way to think about it is that the name of the expression constructor is $(\mathsf{let}\; x = \cdot \;\mathsf{in}\; \cdot)$: it is indexed on some variable x. This corresponds to the Racket notation `(let ([x1 t1])t)`, where the extra set of parentheses around the bindings is to allow for multiple simultaneous bindings, i.e., `(let ([x1 t1][x2 t2] ...)t2)`. As a notational convention, we will sometimes use indentation to mark off the entirety of a let expression's body (the part after in).

Let's write down the rest of the formal semantics of our new language of Boolean and Let expressions (BL). The syntax is as follows:

$$\begin{array}{rcl} b \in \mathbb{B}, & t \in \textsc{Term}, & x \in \textsc{Var} \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & \colorbox{lightgray}{$x \mid \mathsf{let}\ x = t\ \mathsf{in}\ t$} \end{array}$$

To specify our operational semantics, we'll use reduction semantics, which keeps things succinct. You should make sure that you could rephrase these semantics using structural-operational or big-step semantics as well.

We start with the necessary additional syntactic notions: values, redexes, and evaluation contexts:

$$\begin{array}{rcl} v \in \textsc{Value}, & r \in \textsc{Redex}, & E \in \textsc{ECtxt}, \\ v & ::= & b \\ E & ::= & \square \mid E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid \colorbox{lightgray}{$E[\mathsf{let}\ x = \square\ \mathsf{in}\ t]$} \\ r & ::= & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \colorbox{lightgray}{$\mathsf{let}\ x = v\ \mathsf{in}\ t$} \end{array}$$

You can see from the evaluation context $E[\mathsf{let}\ x = \square\ \mathsf{in}\ t]$ that the semantics will evaluate the *bound* expression before it processes the *body* of the let expression. This is consistent with $\mathsf{let}\ x = v\ \mathsf{in}\ t$ as a Redex. Notice that this language introduces no new Values to the language, just a form of named abstraction.

The notions of reduction for the language follow.

$$\begin{array}{c} \rightsquigarrow \subseteq \textsc{Redex} \times \textsc{Term} \\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightsquigarrow t_2 \\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightsquigarrow t_3 \\ \colorbox{lightgray}{$\mathsf{let}\ x = v\ \mathsf{in}\ t$} \rightsquigarrow \colorbox{lightgray}{$[x \Mapsto v]t$} \end{array}$$

The only new notion of reduction is for, yup you guessed it, let! Once you have a value in the binding position, the semantics substitutes it into the body of the expression. A side-warning: we're not quite done yet, because we haven't said anything about how to naively substitute into a let expression, i.e. $[x \Mapsto v]\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ t_2$. We'll cover that in the next section.

Forging ahead, let's define our evaluator. In the past, we allowed *any* Term to count as a program, but here we are going to make a restriction. In particular, we will not allow a program to have any references to variables that have not been bound in the surrounding expression. So an expression like: $\mathsf{let\ x = true\ in\ y}$ doesn't count as a program, because our operational semantics will step this to y, and then we're stuck: there is no notion of reduction for variable references. Instead, the variable should have been replaced by some Value by the time we get to its position in the program. If we don't have a value for our variable when we reference it, then how can we sensibly proceed?!?[6] To capture this, we must introduce a notion of *free variables*. The free variables of an expression are the variables that are referenced in a spot where no surrounding let binding provides their value. We express this idea in precise form as a function:

$$\begin{array}{rcl} FV : \textsc{Term} \to \mathcal{P}(\textsc{Var}) & & \\ FV(\mathsf{true}) & = & \emptyset \\ FV(\mathsf{false}) & = & \emptyset \\ FV(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) & = & FV(t_1) \cup FV(t_2) \cup FV(t_3) \\ FV(x) & = & \{\, x \,\} \\ FV(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) & = & FV(t_1) \cup (FV(t_2) \setminus \{\, x \,\}) \end{array}$$

[6] Emphasis here is on *sensibly*. I'm looking at you C and JavaScript programming languages! The Python programming language allows an unbound variable reference as part of a program, and only fails if that reference is evaluated, producing a "NameError".

As you can see, a let binding handles all free references to the bound variable within it's body. Now we can explicitly define our programs as the set of *closed* terms: terms that have no free variables:

$$\textsc{ClosedTerm} = \{\, t \in \textsc{Term} \mid FV(t) = \emptyset \,\}$$

$$\begin{aligned}
&\textsc{Pgm} = \textsc{ClosedTerm}, \quad \textsc{Obs} = \textsc{Value} \\
&eval : \textsc{Pgm} \to \textsc{Obs} \\
&eval(t) = b \text{ if } t \longrightarrow^{*} b
\end{aligned}$$

Our language definition is almost complete. We just have to extend substitution to handle let bindings. For our immediate purposes, this will be straightforward, but we'll find that extending it to the general case raises some issues.

18.3 Naïve Substitution into Let Bindings

Since our language allows us to put let expressions anywhere, we need to be able to handle them in the bodies of one another. A couple of examples follow:

$$\begin{aligned}
&(1) \quad \mathsf{let\ x = 7\ in\ let\ y = 6\ in\ x} \\
&(2) \quad \mathsf{let\ x = 7\ in\ let\ x = 6\ in\ x}
\end{aligned}$$

If we apply the reduction semantics to the first example above, then we expect to get:

$$\begin{aligned}
&\mathsf{let\ x = 7\ in\ let\ y = 6\ in\ x} \\
\longrightarrow\ &[\mathsf{x} \Mapsto \mathsf{7}]\mathsf{let\ y = 6\ in\ x} \\
=\ &\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let\ y = 6\ in\ x}
\end{aligned}$$

Similarly for the second example we get:

$$\begin{aligned}
&\mathsf{let\ x = 7\ in\ let\ x = 6\ in\ x} \\
\longrightarrow\ &[\mathsf{x} \Mapsto \mathsf{7}]\mathsf{let\ x = 6\ in\ x} \\
=\ &\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let\ x = 6\ in\ x}
\end{aligned}$$

So what should $\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let\ y = 6\ in\ x}$ be? Well, y doesn't really have anything to do with x, so the obvious thing to do is substitute for x in the body of the let. In the case of $\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let\ x = 6\ in\ x}$ though, it sure looks like x should become $\mathsf{6}$ once the inner let expression is evaluated, so the proper behaviour seems to be to let it alone.[7]

More generally, we expect naïve substitution to satisfy the following equations:

$$\begin{aligned}
\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let\ x} = t_1\ \mathsf{in}\ t_2 &= \mathsf{let\ x} = \underline{[\mathsf{x} \Mapsto \mathsf{7}]}t_1\ \mathsf{in}\ t_2 \\
\underline{[\mathsf{x} \Mapsto \mathsf{7}]}\mathsf{let}\ x_0 = t_1\ \mathsf{in}\ t_2 &= \mathsf{let}\ x_0 = \underline{[\mathsf{x} \Mapsto \mathsf{7}]}t_1\ \mathsf{in}\ \underline{[\mathsf{x} \Mapsto \mathsf{7}]}t_2 \quad x_0 \neq \mathsf{x}
\end{aligned}$$

In short, it should not mess with inner bindings of the same variable. In both cases we substitute into the expression that is to be bound to the variable though.

At this point, we might feel satisfied with our equations, but they introduce a bit of gum into the works. If we were to extend our Principle of Definition by Recursion following our pattern to date, we would simply add an extra function:

$$H_{let} : \textsc{Var} \times S \times S \to S$$

And the unique function F would satisfy the equation:

$$F(\mathsf{let}\ x_0 = t_1\ \mathsf{in}\ t_2) = H_{let}(x_0, F(t_1), F(t_2)).$$

[7] Pun not intended, seriously!

This raises a problem: there is no way to define H_{let} to satisfy our equations! By the time we know that $x_0 = \mathsf{x}$, the body of the expression has already been substituted into. Shucks!

Never fear though: it turns out that our principle of definition by recursion can be generalized. The simple one that we used, which proceeds by *structural recursion*, discards the original terms. But that was never strictly necessary: it just happens to be a special case that fits the bill in the grand majority of cases. We introduced that version so as to not make things too complex from the start.

To handle our notion of naïve substitution, we need a stronger principle of recursion.

Proposition 70 (Principle of Definition by Primitive Recursion on $t \in \textsc{Term}$)**.** *Let S be a set and $s_t, s_f \in S$ be two elements,*

$$
\begin{aligned}
H_{if} &: \textsc{Term} \times \textsc{Term} \times \textsc{Term} \times S \times S \times S \to S \\
H_{let} &: \textsc{Var} \times \textsc{Term} \times \textsc{Term} \times S \times S \to S \\
H_{var} &: \textsc{Var} \to S
\end{aligned}
$$

be functions.

Then there exists a unique function

$$F : \textsc{Term} \to S$$

such that

1. $F(\mathit{true}) = s_t$*;*
2. $F(\mathit{false}) = s_f$*;*
3. $F(x) = H_{var}(x)$*.*
4. $F(\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3) = H_{if}(t_1, t_2, t_3, F(t_1), F(t_2), F(t_3))$*;*
5. $F(\mathit{let}\ x = t_1\ \mathit{in}\ t_2) = H_{let}(x, t_1, t_2, F(t_1), F(t_2))$*.*

Proof. Exercise for the reader. □

The principle of definition by primitive recursion provides enough additional structure to enable our component H functions to consider the original terms.

Armed with the principle of definition by *primitive* recursion, we can easily provide a case for let that lets us define the $\underline{[\mathsf{x} \Rrightarrow 7]}$ function we want:

$$
\begin{aligned}
H_{let}(\mathsf{x}, t_1, t_2, t_1', t_2') &= \mathsf{let}\ \mathsf{x} = t_1'\ \mathsf{in}\ t_2 \\
H_{let}(x_0, t_1, t_2, t_1', t_2') &= \mathsf{let}\ x_0 = t_1'\ \mathsf{in}\ t_2' \quad x_0 \neq \mathsf{x}.
\end{aligned}
$$

The main lesson here is that we should be careful with function definitions, and the full-stack semanticist should really know which recursion principle can be used to justify them. Before too long we'll learn the recursion principle to rule all recursion principles (it's called *well-founded recursion*). With that one, we'll even be able to directly define the general substitution function $[\cdot \Rrightarrow \cdot]\cdot$. Sit tight!

18.4 By-Name Let Binding

At this point we have a perfectly fine language definition. But let's consider a slightly different point in the design space. When we defined our language above, we made a design decision without much fanfare: that the expression bound to a let is evaluated to a value before it is substituted. Most popular programming languages operate this way, but it's not the only option. Another option is to substitute *the entire expression* and evaluate it at every variable reference. This is sometimes called *call-by-name*, which is a little odd here because we have no functions to call. The "by-name" part goes back to the design of the Algol 60 language, and the idea was that the "name" was the entire expression. Our original design is called "by-value" since variables are bound only to the values of expressions. Let's consider what changes we would need to make to our semantics to support by-name let bindings. The main changes happen in the syntax for our extra forms:

$$\begin{array}{rcl} E & ::= & \Box \mid E[\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t] \\ r & ::= & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x = \boxed{t}\ \mathsf{in}\ t \end{array}$$

First, we remove the evaluation context for let because we will no longer evaluate the bound expression. Then the redexes change in that any arbitrary let expression counts as a redex. Thus, our notion of reduction for let is updated.

$$\boxed{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2} \rightsquigarrow \boxed{[x \Rrightarrow t_1]t_2}$$

The significance of this style of application is that it may yield *worse* performance lead you to perform more work if you refer to the same variable many times, as in

$$\mathsf{let\ x} = \mathit{BIGEXPRESSION}\ \mathsf{in\ if\ x\ then\ x\ else\ x}$$

Here *BIGEXPRESSION* gets recomputed 3 times. Under by-value it would only be computed once.

On the other hand, by-name can be more efficient if it avoids unnecessary work., for example:

$$\mathsf{let\ x} = \mathit{BIGEXPRESSION}\ \mathsf{in\ if\ true\ then\ false\ else\ true}$$

Under the by-name model, *BIGEXPRESSION* never gets run at all, while it gets run in by-name once.

Later we will be introduced to a computation model called "by-need" that tries to blend both by-value and by-name to get the best of both worlds (though it still costs you some things). In practice, production languages[8] always use "by-need" to mimic by-name behaviour, but understanding by-name semantics is typically sufficient to reason about such languages, because they avoid features that would let you tell the difference.

18.5 Capture-Avoiding Substitution

In the semanticses above, both by-value and by-name, we take advantage of a significant property of programs: that we only substitute closed expressions into the bodies of let expressions. To see this, consider that we only consider closed expressions to be programs, and that furthermore, evaluation contexts do not allow us to evaluate under variable bindings, which are the only places where free variables could be found.

So evaluation can be defined using naïve substitution without any trouble, but what happens if we try to substitute a term with free variables into the body of a let. Well, things can go badly.

Consider the following expression:

$$\begin{array}{l} \mathsf{let\ z} = 3 \\ \mathsf{in\ let\ y} = \mathsf{z} \\ \quad \mathsf{in\ let\ z} = 2 \\ \quad\quad \mathsf{in\ y} \end{array}$$

and suppose that we can perform the reductions in any order. If we start with the outermost and work our way in, then everything goes fine, since it corresponds to our by-value evaluation which yields 3. But suppose we reduce the *middle* binding first.

$$\begin{array}{lcl} \mathsf{let\ z} = 3 & \longrightarrow & \mathsf{let\ z} = 3 \\ \mathsf{in\ let\ y} = \mathsf{z} & & \mathsf{in\ let\ z} = 2 \\ \quad \mathsf{in\ let\ z} = 2 & & \quad \mathsf{in\ z} \\ \quad\quad \mathsf{in\ y} & & \end{array}$$

Then no matter the order of the last two steps, we end up with 2 as our result. What gives? Well, the z that is bound to y is meant to refer to the binding of z on the outside, but by willy-nilly substituting z

[8] The Haskell programming language is the most well-known of this sort.

under another binding of the same variable, we *inadvertently capture* z. How can we avoid this? Well, one way is to *rename* the inner z binding to some other innocuous name, say g.

```
let z = 3              ⟶    let z = 3
in let y = z                in let g = 2
   in let g = 2                in z
      in y
```

Now that we have renamed the inner z to g, we have nothing to worry about. So the lesson here is that substituting open terms into the body of a let must be careful to avoid inadvertent capture.

We can develop a richer notion of substitution that exactly addresses this issue. This new operation is called *capture-avoiding* substitution, and it is the standard notion of substitution.

First off, why would we want to do this rewrite out of order? The most important answer is that we would like to understand what kinds of changes we can make to a program that will preserve the result of *eval*. These are called *correctness-preserving program transformations*. It turns out that applying notions of reduction out of order are often correctness-preserving, and doing so can often improve the performance of a program. I bet that as a programmer you have essentially made these kinds of rewrites to your code before, based on the intuition that the program will still "do the same thing". We would like to be able to formalize this kind of thing, and being able to perform substitutions under variable binders is a common example.[9]

So, we can see that naïve substitution doesn't cut it in this case. That means that we need to develop a new kind of substitution that supports substituting terms with free variables into other terms. We pursue this now.

18.5.1 α-equivalence, or, Bound Variable Names Don't Matter

One of the key observations above that helped us fix our errant instance of substitution is that the choice of bound variable names shouldn't really matter in a program. For instance, the following two expressions:

```
let x = 7 in x
let y = 7 in y
```

Are for all intents and purposes the same. While useful variable names are helpful to humans when they want to understand programmer intent, the language semantics just doesn't care which variable names you choose, so long as the variable references line up with the variable bindings in the same way.

The alternative is a programming language where you have to worry all the time about what local variable names are used within each function. That makes it hard to build correct program modules independently: every piece of code has to watch out for colliding with variable names from other modules...that's almost as bad as using global variable names everywhere!

In the language we have described so far this property that bound variable names don't matter holds, and we can formalize it as an *equivalence relation*, using naïve substitution to help us.

Definition 8 (Alpha Equivalence)**.** *Let $\sim_a \subseteq$ TERM $\times$ TERM be defined by the following rules:*

$$\frac{}{x \sim_a x} \qquad \frac{}{b \sim_a b} \qquad \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22} \quad t_{13} \sim_a t_{23}}{\textit{if}\ t_{11}\ \textit{then}\ t_{12}\ \textit{else}\ t_{13} \sim_a \textit{if}\ t_{21}\ \textit{then}\ t_{22}\ \textit{else}\ t_{23}}$$

$$\frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{\textit{let}\ x = t_{11}\ \textit{in}\ t_{12} \sim_a \textit{let}\ x = t_{21}\ \textit{in}\ t_{22}}$$

$$\frac{\textit{let}\ x_3 = t_{11}\ \textit{in}\ [x_1 \Rrightarrow x_3]t_{12} \sim_a \textit{let}\ x_3 = t_{21}\ \textit{in}\ [x_2 \Rrightarrow x_3]t_{22}}{\textit{let}\ x_1 = t_{11}\ \textit{in}\ t_{12} \sim_a \textit{let}\ x_2 = t_{21}\ \textit{in}\ t_{22}} \quad \begin{array}{l} x_1 \neq x_2, \\ x_3 \notin FV(t_{12}) \cup FV(t_{22}) \end{array}$$

Alpha-equivalence, written $\sim_a$, is a binary relation on terms. The "alpha" bit is a painfully cute reference to "alphabet", implying that the programs are equivalent up to the choice of names for bound variables.

[9]A refactoring tool can break your code if it is not sufficiently careful with performing program rewrites underneath variable bindings!

Almost all of the cases that define this relation look exactly like equality: If we throw out the last rule, then the definition *is* exactly equality. This makes it clear that identical terms are alpha-convertible. The last rule, though, is the only interesting one. Essentially it says that two let bindings with *different* bound variable names (i.e., $x_1 \neq x_2$) are alpha-convertible if *renaming their bound variables* to a common name that does not capture any previously-free variables (i.e., $x_3 \notin FV(t_{12}) \cup FV(t_{22})$), suffices to make the terms alpha-equivalent. Later you may see alternative definitions of the same notion of alpha-equivalence.

Now, when you look at the definition of $\sim_a$, it "relates" terms by reconciling the variable names bound by let expressions, but we are technically on the hook to confirm that this induces some notion of equivalence, i.e. that two terms are "equal enough for our purposes." In particular, we should prove that $\sim_a$ is in fact an *equivalence relation*.

Proposition 71 ($\sim_a$ is an equivalence relation)**.** *Given $t_1, t_2, t_3 \in$ Term, the following are true:*

1. $t_1 \sim_a t_1$;
2. *If* $t_1 \sim_a t_2$ *then* $t_2 \sim_a t_1$;
3. *If* $t_1 \sim_a t_2$ *and* $t_2 \sim_a t_3$ *then* $t_1 \sim_a t_3$.

Equivalence relations arise *all the time* in mathematics. They are a mathematical way of characterizing groups of stuff that are "like one another" (i.e. are equivalent) so long as you ignore some uninteresting differences. *Equality* (or *identity*) = is also an equivalence relation, but it is the one that ignores nothing: two entities are only equal if they are truly identical: one and the same. In our case here, two Terms are alpha-equivalent if they are the same except for the choice of variables bound by let. Our hope is that our choices for bound variable names should not "matter," in the sense that it should not affect the meaning of a program, though it can affect whether some programmer understands what you are doing. As it turns out, we can prove that names don't matter (at least not to evaluation).

Proposition 72. *If $t_1 \sim_a t_2$ then $eval(t_1) = eval(t_2)$.*

Proof. An exercise. □

Thus, α-equivalence is an example of a correctness-preserving transformation on programs, or what the kids might call "a legal refactoring".

18.6 Generalizing Substitution

We just saw that evaluation, whose definition depends under the hood on naïve substitution, is invariant with respect to alpha equivalence. However, naïve substitution itself *does not* respect alpha equivalence. We can see this by considering the underlying problem with one of our earlier examples:

$$z \sim_a z$$

and

$$\begin{array}{l}\mathsf{let\ z = 2}\\ \mathsf{in\ y}\end{array} \sim_a \begin{array}{l}\mathsf{let\ g = 2}\\ \mathsf{in\ y}\end{array}$$

but

$$[\mathsf{y} \mapsto \mathsf{z}]\left(\begin{array}{l}\mathsf{let\ z = 2}\\ \mathsf{in\ y}\end{array}\right) = \begin{array}{l}\mathsf{let\ z = 2}\\ \mathsf{in\ z}\end{array} \not\sim_a \begin{array}{l}\mathsf{let\ g = 2}\\ \mathsf{in\ z}\end{array} = [\mathsf{y} \mapsto \mathsf{z}]\left(\begin{array}{l}\mathsf{let\ g = 2}\\ \mathsf{in\ y}\end{array}\right)$$

So as far as naive substitution work, bound variable names matter! We didn't run into problems earlier because our operational semantics only substitutes *closed terms*, a special case that causes no problems.

Proposition 73. *If t_{11} is closed, $t_{11} \sim_a t_{21}$, and $t_{12} \sim_a t_{22}$, then $[x \mapsto t_{11}]t_{12} \sim_a [x \mapsto t_{21}]t_{22}$.*

However, naïvely substituting *open terms*—terms that have free variables—can cause problems, as above.

The key to making substitution work for *arbitrary* terms, with or without free variables, rests with alpha-equivalence. We've seen already that we can substitute open terms without incident if we carefully rename bound variables along the way. We can bake this technique directly into an enhanced version of substitution, and the resulting operation will respect alpha equivalence, as we desire.

In essence, these propositions say that if two programs differ only in their choice of bound variables, then their results also only differ in their bound variable names. Let's formalize this with a new function definition:

$$
\begin{aligned}
&\underline{[t/x]} : \textsc{Term} \to \textsc{Term}\\
\underline{[t/x]}\mathsf{true} &= \mathsf{true}\\
\underline{[t/x]}\mathsf{false} &= \mathsf{false}\\
\underline{[t/x]}(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= \mathsf{if}\ \underline{[t/x]}t_1\ \mathsf{then}\ \underline{[t/x]}t_2\ \mathsf{else}\ \underline{[t/x]}t_3\\
\underline{[t/x]}x &= t\\
\underline{[t/x]}x_0 &= x_0 \text{ if } x \neq x_0\\
\underline{[t/x]}(\mathsf{let}\ x_0 = t_1\ \mathsf{in}\ t_2) &= \underline{[t/x]}(\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ [x_0 \Rightarrow x_1]t_2) \quad x_0 \in \{\, x \,\} \cup FV(t)\\
\underline{[t/x]}(\mathsf{let}\ x_0 = t_1\ \mathsf{in}\ t_2) &= \mathsf{let}\ x_0 = \underline{[t/x]}t_1\ \mathsf{in}\ \underline{[t/x]}t_2 \quad x_0 \notin \{\, x \,\} \cup FV(t)\\
&\text{where } x_1 \text{ is the } \textit{least} \text{ variable such that } x_1 \notin \{\, x \,\} \cup FV(t) \cup FV(\mathsf{let}\ x_0 = t_1\ \mathsf{in}\ t_2)
\end{aligned}
$$

The key equations here are the last two. Notice how the second-to-last uses naïve substitution to rename the bound variable of the let expression to a new variable that does not already appear free. Mind you, we could be more conservative about renaming, only doing so where *strictly* necessary, so as to keep the original variable names wherever possible. Doing so would be very helpful for a human-facing implementation, but here we are primarily interested in the mathematics: *bound variable names don't matter!*

Given this definition,we can produce our final function $[\cdot/\cdot]\cdot : \textsc{Term} \times \textsc{Var} \times \textsc{Term} \to \textsc{Term}$.

We call this *capture-avoiding substitution* because it never accidentally captures a bound variable.

There are a few things worth mentioning about this particular definition. In the literature, a number of subtle points are typically glossed over because you can get pretty far with approximately the right definition, but it's good to know the finer details so that they don't come back to bite you.

Notice at the end the side condition about *least* variables. This function assumes that variables have some ordering $x_0, x_1, x_2, \ldots$ to make sure that the resulting function is indeed a function, that is, always maps its input term to a unique output. Yes, this is super-contrived, but to be honest there are some programming language implementations that essentially do something like this to deterministically choose new variable names.

Confirming that the equations above, as written, take a little work. In short, we can fuse the last two equations (by properly coalescing their side-conditions), and then appeal to the principle of definition by primitive recursion to justify this function definition. Then we can define our general ternary substitution function. We still cannot define capture-avoiding substitution in general from the start, since it's technically not defined by recursion over the structure of the last term.[10].

Capture-avoiding substitution satisfies the properties that we care about:

Proposition 74. *If t is closed then $[x \Rightarrow t]t_1 \sim_a [t/x]t_1$.*

Thus, capture-avoiding substitution agrees *enough* with naïve substitution for closed terms, so the latter can be used to define an evaluator function without breaking anything.

Proposition 75. *If $t_{11} \sim_a t_{21}$ and $t_{12} \sim_a t_{22}$ then $[t_{11}/x]t_{12} \sim_a [t_{21}/x]t_{22}$.*

Proof. By induction over $t_{11} \sim_a t_{22}$.[11] □

This proposition tells us that unlike naïve substitution, capture-avoiding substitution plays well with alpha-equivalence. So now renaming bound variables doesn't really matter (as long as you continue not to

[10]...despite what a number of textbooks say (sigh).

[11]I recommend working through this.

care about variable names...it matters a lot to programmers if you're trying to implement a debugger for instance)!

18.7 Observational Equivalence

Throughout the course, we have emphasized that our various operational semanticses are ultimately tools for defining and reasoning about some evaluation function:

$$eval : \textsc{Pgm} \to \textsc{Obs}$$

The operational semantics themselves give us but one tool to reason about this partial function. In particular it suggests to us at least one strategy for actually implementing the language. However, at the end of the day the *eval* partial function is the real semantics of our language, and not every property of that function can be easily deduced from a deterministic operational semantics.

Let's consider as our language the Boolean expressions with let:

$$\begin{array}{lcl} t \in \textsc{Term}, & b \in \mathbb{B}, & x \in \textsc{Var}, \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid x \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \\ b & ::= & \mathsf{true} \mid \mathsf{false} \end{array}$$

And consider the following definitions for programs and observable results.

$$\begin{array}{l} \textsc{Pgm} = \{\, t \in \textsc{Term} \mid FV(t) = \emptyset \,\} \\ \textsc{Obs} = \mathbb{B} \end{array}$$

We will not consider *how eval* is defined, but simply assume that by some means we have established a mapping from programs to observable results.

What kinds of questions can we ask about our language, independent of how its evaluator was defined? One of the most important notions we can consider is *observational equivalence*, which is also sometimes called *contextual equivalence* or *operational equivalence*.

As hinted by the second name, we first need to consider program contexts, in particular *compatible contexts*:

$$\begin{array}{l} C \in \textsc{Ctxt} \\ C ::= \square \mid C[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid C[\mathsf{if}\ t\ \mathsf{then}\ \square\ \mathsf{else}\ t] \mid C[\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ \square] \mid C[\mathsf{let}\ x = \square\ \mathsf{in}\ t] \mid C[\mathsf{let}\ x = t\ \mathsf{in}\ \square] \end{array}$$

Generally, the compatible contexts are formed from the grammar of Terms for the language, allowing a hole in any position where a Term could be. This allows us to formally talk about any context where a term could appear. We use this notion to define observational equivalence.

Definition 9 (Observational Equivalence)**.** *Let $t_1, t_2 \in \textsc{Term}$. Then $t_1 \simeq t_2$ if and only if:*

1. *There is at least one context $C \in \textsc{Ctxt}$ such that $C[t_1], C[t_2] \in \textsc{Pgm}$.*
2. *For any $C \in \textsc{Ctxt}$, $C[t_1] \in \textsc{Pgm}$ if and only if $C[t_2] \in \textsc{Pgm}$;*
3. *$eval(C[t_1]) = eval(C[t_2])$ for all $C \in \textsc{Ctxt}$ such that $C[t_1], C[t_2] \in \textsc{Pgm}$.*

This definition is stated for our language specifically, but the concept can be extended in various ways [Harper, 2012, Part XVIII]. The first criterion for observational equivalence is that there must be *at least one* program where the two terms t_1 and t_2 can be interchanged. If not, then the two terms are simply incomparable. As it turns out, this is always true in our language. To show this, notice that programs are simply closed terms. A simple context that would satisfy this property would collect all of the free variables of t_1 and t_2 and bind those variables to true, i.e. $\mathsf{let}\ x_1 = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ x_2 = \mathsf{true}\ \mathsf{in}\ \ldots\ \mathsf{in}\ []$.

The second criterion makes sure that the two terms are *always* interchangeable. If not, then you could surely distinguish them.

The third criterion is the real payload. It states that two terms are observationally equivalent if they *mean the same thing*, as far as the evaluator is concerned, in any context that accepts both of them. Note that in this case the equality sign between the two evaluations is given a hat to represent *Kleene equality*. Two expressions satisfy Kleene equality if and only if both expressions are defined and equal or both are undefined. So Kleene equality deals with the possibility that the *eval* expression may be undefined on both sides. If one side is undefined and the other has a value, then they are not Kleene equal.

This definition should seem relatively intuitive. As far as our black-box evaluator is concerned, terms are observationally equivalent if the evaluator cannot "observe" any difference between their behaviour.

This notion is quite useful to programmers and language implementers. It can be used to justify program optimizations, program refactorings, and a host of other things, all without (technically) focusing on the details of *how* the evaluator was defined. In fact, regression testing is essentially built on this idea: if the expected results of the regression tests change, then something went wrong!

Its worth noting, though, that in general it's hard to prove observational equivalence. If you look at the structure of the definition, it quantifies over all contexts that complete two terms. In general, that's not an easy thing to work with. Usually language theorists find useful ways to approximate observational equivalence: they define some useful equivalence relation $\approx$ that is *sound* with respect to observational equivalence, i.e., if $t_1 \approx t_2$ then $t_1 \simeq t_2$, but probably not vice-versa.

Bibliography

P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

J. Avigad. Reliability of mathematical inference. *Synthese*, 2020. doi: 10.1007/s11229-019-02524-y. `https://doi.org/10.1007/s11229-019-02524-y`.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

J. Bagaria. Set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, winter 2014 edition, 2014. URL `http://plato.stanford.edu/archives/win2014/entries/set-theory/`.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `http://doi.acm.org/10.1145/3182657`.

H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

A. Bauer. Proof of negation and proof by contradiction. Mathematics and Computation Blog, March 2010. `http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/`.

L. Crosilla. Set Theory: Constructive and Intuitionistic ZF. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(89)90069-8. URL `http://www.sciencedirect.com/science/article/pii/0304397589900698`.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 1st edition, 2009.

J. Ferreirós. The early development of set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. URL `https://plato.stanford.edu/archives/sum2019/entries/settheory-early/`.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

P. R. Halmos. *Naive Set Theory*. Springer-Verlag, first edition, Jan. 1960. ISBN 0387900926.
A classic introductory textbook on set theory.

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. URL `http://www.cs.cmu.edu/%7Erwh/plbook/1sted-revised.pdf`.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124 (2):103–112, Feb. 1996. ISSN 0890-5401.

G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.

R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015. URL https://robbertkrebbers.nl/thesis.html.

C. Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1): 161–171, 1921. doi: 10.4064/fm-2-1-161-171. https://web.archive.org/web/20190429103938/http://matwbn.icm.edu.pl/ksiazki/fm/fm2/fm2122.pdf.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi: 10.1093/comjnl/6.4.308. URL https://doi.org/10.1093/comjnl/6.4.308.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, Feb. 2009. ISSN 0890-5401.

P. Maddy. Believing the axioms. I. *The Journal of Symbolic Logic*, 53(02):481–511, 1988.
An interesting (though complicated) analysis of why set theorists believe in their axioms.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

J. H. J. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Feb. 1969. URL http://hdl.handle.net/1721.1/64850.

A. M. Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004a. doi: 10.1016/j.jlap.2004.03.009. URL http://dx.doi.org/10.1016/j.jlap.2004.03.009.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.

B. Popik. "pull yourself up by your bootstraps". Weblog entry, September 2012. https://www.barrypopik.com/index.php/new_york_city/entry/pull_yourself_up_by_your_bootstraps/.

E. Reck and G. Schiemer. Structuralism in the Philosophy of Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2020 edition, 2020.

D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4): 15:1–15:41, May 2009. ISSN 0164-0925.

W. Sieg and D. Schlimm. Dedekind's analysis of number: Systems and axioms. *Synthese*, 147(1):121–170, Oct 2005.

J. Spolsky. The law of leaky abstractions. Joel on Software Blog, November 2002. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

G. L. Steele. Debunking the "expensive procedure call"" myth or, procedure call implementations considered harmful or, lamdba: The ultimate goto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. URL http://dspace.mit.edu/handle/1721.1/5753.

S. Stenlund. Descriptions in intuitionistic logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 197 – 212. Elsevier, 1975. URL http://www.sciencedirect.com/science/article/pii/S0049237X08707328.

M. Tiles. Book Review: Stephen Pollard. Philosophical Introduction to Set Theory. *Notre Dame Journal of Formal Logic*, 32(1):161–166, 1990.
A brief introduction to the philosophical issues underlying set theory as a foundation for mathematics.

C. Urban and M. Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '05, page 25–32, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930728. doi: 10.1145/1088454.1088458. URL `https://doi.org/10.1145/1088454.1088458`.

C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, page 35–50, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540735946. doi: 10.1007/978-3-540-73595-3_4. URL `https://doi.org/10.1007/978-3-540-73595-3_4`.

D. van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994. ISBN 978-3-540-57839-0.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.

B. Zimmer. figurative "bootstraps". email to linguistlist mailing list, August 2005. `http://listserv.linguistlist.org/pipermail/ads-l/2005-August/052756.html`.