1 Introduction

Lexically-scoped variables are great for programming, but a source of headaches for theorists. For the programmer, lexical scoping ensures that their choice of local bound variable names has no affect on the meaning of the rest of a program. This prevents strange forms of “spooky action at a distance” where some unfortunate confluence of names chosen in two parts of a software system cause chaos. For the theorist, however, lexical scoping imposes the obligation to ensure that bound variable names don’t matter. This is most easily seen in the definition of capture-avoiding substitution, which must jump through some hoops to avoid the problems that naïve substitution might cause.

In [Barendregt, 1985], the logician Hendrik “Henk” Barendregt, proposes a clever, intuitive, and now-ubiquitous approach to getting almost the best of both worlds: lexical scoping for programmers with minimal extra hoops for the theorist. He summarizes his approach as follows:

1. Identify two terms if each can be transformed to the other by a renaming of its bound variables.
2. Consider a λ-term as a representative of its equivalence class.
3. Interpret substitution \[ t/x \] \( t \) as an operation on the equivalence classes of \( t \) and \( t_0 \). This operation can be performed using representatives, provided that the bound variables are named properly as formulated in the variable convention below.

Barendregt goes on to state two conventions that are used throughout the book:

1. Convention 2.1.12: Terms that are α-congruent are identified.
2. Variable Convention 2.1.13: If \( t_1, \ldots, t_n \) occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Following an example, there is stated a:

---

1. Convention 2.1.12: Terms that are α-congruent are identified.
2. Variable Convention 2.1.13: If \( t_1, \ldots, t_n \) occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

---

\( \alpha \)-equivalence.
1. Moral 2.1.14: Using conventions 2.1.12 and 2.1.13, one can work with \( \lambda \)-terms in the naïve way.

Naïve means that substitution and other operations on terms can be performed without questioning whether they are allowed.

Conventions 2.1.12 and 2.1.13 are often referred to as Barendregt’s variable convention. Many PL papers in the literature include an off-hand remark roughly of the form: “Following Barendregt, we identify terms up to choice of bound variables.” While the general outline of this idea seems pretty straightforward and easy, the formal implications, featuring a few unexpected traps and pitfalls, which may arise when applied without care, are definitely not. Rendering it formally, and extending it beyond capture-avoiding substitution, takes some work. Some work in the context of automating the mechanization of an analogous convention in the Isabelle/HOL proof assistant, goes so far as to critique Barendregt’s conventions as being insufficient even for some purposes to which it is applied in the monograph [Urban and Norrish, 2005, Urban et al., 2007]. Teaching a computer to automatically identify bound variable names exposed some gaps, which also arise when working rigorously on paper.

Here we treat Barendregt’s conventions quite formally, so as to expose the subtleties and proof obligations in high relief. This note shows in gory detail just how much machinery such seemingly off-hand statements evoke, involving quite a few definitions, propositions, and proof obligations. Most uses of this strategy do not go into quite so much detail about what’s involved in “identifying terms up to ...”, but I think that seeing the details can help you:

1. precisely formalize these conventions, and other related notions of identifying terms up to some ignorable detail. By knowing how the precise details work, you can often see things that need more care than might otherwise be expected;

2. appreciate that doing this formally yields a new notion of syntax, which fundamentally involves an interplay between a quite abstract notion of syntax—even more abstract than syntax trees—and more concrete representations of that syntax—whose role is played by our previous abstraction of syntax trees. In short, this approach builds a new expressive abstraction for syntax atop your old one;

3. draw out important implications for computer implementations of algorithms and interpreters for semantics based on syntax “modulo choice of bound variables.” The key takeaway is that implementations have a lot of flexibility. These programs operate on representations of syntax objects, in that they may choose bound variable names any way they wish. In a very real sense, choice of names no longer matters.

4. identify where and how misunderstandings can lead to broken definitions, theorems, and proofs, and why in practice this approach can sometimes be confusing. As we see below, working this way typically involves the use of standard concrete term notation as a stand-in for abstract operators over equivalence classes of concrete terms that in reality are denoted using a combination of concrete terms, injection operations, and abstract operators over sets of terms. The concrete term notation more directly conveys the essence of definitions and formal developments, but to some extent it requires you to “read between the notation”, which is easier to do if you know what precisely the missing notation would look like.

Used well, this machinery provides a useful abstraction that quite admirably dispatches concerns about the names of bound variables.

This exposition also intends to convey a more abstract meta-lesson. When it comes to set-theoretic mathematics of syntax and semantics, “concrete” and “abstract” are merely relative notions. Previously we appealed to syntax trees to dispatch the headaches caused by consideration of syntax as strings of characters. Here, we address additional facets of syntax that cause headaches for semantics and other metatheory, by adding another layer on top of our representations of terms as trees. In doing so, our old abstract syntax trees become for our purposes too concrete, and our new notion of abstract syntax exhibits some nice new properties. In general, access to the full power of set theory enables us to build up useful
abstractions that encapsulate messy details so that we can ignore them. Just as we can build beautiful clean
precise programming abstractions atop X86, so can we do mathematical counterpart atop set theory.

2 Syntax

We begin with syntax in the style we’ve typically followed. Here, though, we’ll call it “concrete” to allude to
the idea that it exposes too many implementation details, in particular the choice of bound variable names.

Definition 1 (Concrete Syntax).

\[ x \in \text{VAR}, \quad n \in \mathbb{Z}, \quad t \in \text{CONCRETE\textsc{Term}} \]

\[ t ::= \quad n \mid t + t \mid x \mid \lambda x.t \mid tt \mid \text{let } x = t \text{ in } t \]

Through the rest of these notes, we build up a notion of “abstract syntax”, and use that to define our
language. We’ll find that the relationship between our abstract syntax and an implementation, for example,
becomes a bit more remote than before, but also opens up more options for faithfully encoding our syntax
as data structures. That’s kind of the point: we want to abstract away from details that should not constrain
the implementor.

To define our abstract syntax, we need some relation and function definitions. There is more than one set
of definitions that can get us ultimately to the same place (i.e., Defn. 3), each exposing different properties
of the final product. Consider the path that we take in the following to be somewhat arbitrary. Somewhat.

To start, we establish a few standard notions related to variables: free variables, occurring variables, and
naïve substitution, and capture-avoiding substitution. We exploit each of these to construct our new ab-
stract syntax, called \( \text{TERM} \), but we also use some of them to illustrate differences between \( \text{CONCRETE\textsc{Term}} \)
and this new notion, by developing analogous notions in some cases, or explaining why no analogous
notion exists in other cases.

Definition 2 (Free Variables).

\[ FV : \text{CONCRETE\textsc{Term}} \rightarrow \mathcal{P}(\text{VAR}) \]

\[ FV(n) = \emptyset \]

\[ FV(t_1 + t_2) = FV(t_1) \cup FV(t_2) \]

\[ FV(x) = \{ x \} \]

\[ FV(\lambda x.t) = FV(t) \setminus \{ x \} \]

\[ FV(t_1 t_2) = FV(t_1) \cup FV(t_2) \]

\[ FV(\text{let } x = t_1 \text{ in } t_2) = FV(t_1) \cup (FV(t_2) \setminus \{ x \}) \]

Definition 3 (Occurring Variables (i.e. Free or Bound Variables)).

\[ Vars : \text{CONCRETE\textsc{Term}} \rightarrow \mathcal{P}(\text{VAR}) \]

\[ Vars(n) = \emptyset \]

\[ Vars(t_1 + t_2) = Vars(t_1) \cup Vars(t_2) \]

\[ Vars(x) = \{ x \} \]

\[ Vars(\lambda x.t) = \{ x \} \cup Vars(t) \]

\[ Vars(t_1 t_2) = Vars(t_1) \cup Vars(t_2) \]

\[ Vars(\text{let } x = t_1 \text{ in } t_2) = Vars(t_1) \cup Vars(t_2) \]
Definition 4 (Naïve Substitution).

\[ [x \mapsto t] : \text{CONCRETE\textsc{Term}} \to \text{CONCRETE\textsc{Term}} \]

\[ [x \mapsto t] n = n \]

\[ [x \mapsto t](t_1 + t_2) = ([x \mapsto t]t_1) + ([x \mapsto t]t_2) \]

\[ [x \mapsto t]x = t \]

\[ [x \mapsto t]x_0 = x_0 \text{ if } x \neq x_0 \]

\[ [x \mapsto t]\lambda x.t = \lambda x.t \]

\[ [x \mapsto t]\lambda x_0.t_0 = \lambda x_0.[x \mapsto t]t_0 \text{ if } x \neq x_0 \]

\[ [x \mapsto t](t_1 t_2) = ([x \mapsto t]t_1)([x \mapsto t]t_2) \]

\[ [x \mapsto t](\text{let } x = t_1 \text{ in } t_2) = \text{let } x = [x \mapsto t]t_1 \text{ in } t_2 \]

\[ [x \mapsto t](\text{let } x_0 = t_1 \text{ in } t_2) = \text{let } x_0 = [x \mapsto t]t_1 \text{ in } [x \mapsto t]t_2 \text{ if } x \neq x_0 \]

Recall that naïve substitution is naïve in the sense that it can cause free variables to be captured as a result of substitution. For example, \([y \mapsto x] \lambda x.y = \lambda x.x\). On the other hand, \([y \mapsto x] \lambda y.y = \lambda x.x\). So naïve substitution does not preserve free variables. Capture-avoiding substitution (Defn. 7) solves this problem, but by adding both annoying and excessively specific complexity to its definition.

Now using the above notions, we introduce alpha equivalence, a relationship between CONCRETE\textsc{Terms} which says that they are identical but for their choice of bound variables. For many purposes, we consider such CONCRETE\textsc{Terms} to be “the same”, even though set theory says otherwise. Our goal here is to rigorously blur this distinction, replacing this notion of equivalence for CONCRETE\textsc{Term} with set-theoretic identity for our soon-to-be-defined set TERM.

Definition 5 (Alphabetic equivalence (a.k.a. Alpha Equivalence)).

\[ \sim_a \subseteq \text{CONCRETE\textsc{Term}} \times \text{CONCRETE\textsc{Term}} \]

\[ \frac{t_1 \sim_a t_2}{t_1 + t_2 \sim_a t_1 + t_2} \quad (\text{oplus}) \quad \frac{x \sim_a x}{(\lambda x.t) \sim_a (\lambda x.t)} \quad (\text{ovar}) \]

\[ \frac{(\lambda x_3.[x_1 \mapsto x_3]t_1) \sim_a (\lambda x_3.[x_2 \mapsto x_3]t_2)}{(\lambda x_1.t_1) \sim_a (\lambda x_2.t_2)} \quad (\text{olam2}) \]

\[ \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{(\text{let } x_3 = t_{11} \text{ in } [x_3]t_{12} \sim_a \text{let } x_3 = t_{21} \text{ in } [x_3]t_{22})} \quad (\text{olet2}) \]

\[ \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{(\text{let } x = t_{11} \text{ in } t_{12} \sim_a \text{let } x = t_{21} \text{ in } t_{22})} \quad (\text{olet1}) \]

The key rules underlying \(\alpha\)-equivalence are (olam2) (olet2): removing them would give an inductive definition of equality. In essence, these two rules say that two lambda abstractions (or let abstractions) with different bound variables are equivalent if renaming their bound variables to some common variable yields equivalent terms. We use naive substitution to specify renaming, and use the \(\text{Vars}\) function to restrict the possible common variable names, so as to avoid variable capture. We could have use capture-avoiding substitution instead of naïve substitution, yielding a slightly different inductive definition of the same relation.

It’s worth taking a moment to ask yourself what that difference would be, and what other changes could be made without breaking the relation.

Of utmost importance to us is that \((\sim_a \cdot)\) is an equivalence relation.
**Proposition 1** ((· ∼_a ·) is an Equivalence Relation).

1. ∀t ∈ CONCRETE_TERM. t ∼_a t. In words, ∼_a is reflexive;
2. ∀t_1, t_2 ∈ CONCRETE_TERM. t_1 ∼_a t_2 ⇒ t_2 ∼_a t_1. In words, ∼_a is symmetric;
3. ∀t_1, t_2, t_3 ∈ CONCRETE_TERM. t_1 ∼_a t_2 ∧ t_2 ∼_a t_3 ⇒ t_1 ∼_a t_3. In words, ∼_a is transitive.

Any equivalence relation can be used to convey the idea that “sure these two things are not necessarily identical as far as mathematics is concerned, but I want to treat them as the same when we ignore certain details.” In many mathematical arguments, we can replace set-theoretic identity (i.e. equality) with some other equivalence relation, which gives us more “slack”. Another approach is to use the equivalence relation to define a new set of sets of equivalent objects. This set is typically called the set of equivalence classes for a particular equivalence relation, and the process of constructing it is, for somewhat archaic reasons that come from abstract algebra, called quotienting the original set with respect to an equivalence relation. Once this is done, we can lift relevant operators over concrete elements to apply to equivalence classes.

In the following, we quotient CONCRETE_TERM, with respect to α-equivalence, and then lift our operations to apply to α-equivalence classes, which become our notion of TERMS that don’t care about bound variable names.

This lets us be rigorous and abstract in our reasoning, and flexible regarding the relationship between our maths and our implementations. Plus it lets us hide distracting details from our subsequent developments and treat those details in a highly permissive and flexible fashion.

We start with the tedium that we later abstract: defining capture-avoiding substitution. We can prove that the equations in Defn. 7 definitely describe a unique function starting at least a couple of different ways.

1. We can inductively define capture-avoiding substitution as a quaternary inductive relation, then prove that this relation is a partial function, is total, and satisfies the equations in Defn. 7.
2. We can define capture-avoiding substitution in terms of naïve substitution, exploiting the principle of recursion on the size of CONCRETE_TERMS. We can then apply equational reasoning to prove that this function satisfies the equations in Defn. 7.

Either way, we have proven the existence of a function that satisfies the equations in Defn. 7, at which point we prove that at most one function satisfies these equations, proceeding by induction on the size of CONCRETE_TERMS.

Either way, we appeal to the size of CONCRETE_TERMS, so here is the relevant function definition, as well as an equational definition of capture-avoiding substitution.

**Definition 6** (Size of a CONCRETE_TERM).

\[
\text{size : CONCRETE_TERM} \rightarrow \mathbb{N} \\
\text{size}(n) = 1 \\
\text{size}(t_1 + t_2) = 1 + \text{size}(t_1) + \text{size}(t_2) \\
\text{size}(x) = 1 \\
\text{size}(\lambda x.t) = 1 + \text{size}(t) \\
\text{size}(t_1 t_2) = 1 + \text{size}(t_1) + \text{size}(t_2) \\
\text{size(\text{let } x = t_1 \text { in } t_2) = 1 + \text{size}(t_1) + \text{size}(t_2)}
\]

\[7\] Neither of these is as straightforward as is often claimed in texts. In particular, it is not defined by structural recursion on terms!
Definition 7 (Capture-Avoiding Substitution).

\[
\ell\{b, t\} \times \text{VAR} \times \text{CONCRETE}\text{TERM} \rightarrow \text{CONCRETE}\text{TERM}
\]

\[
[t/x]b = b
\]

\[
[t/x]x = t
\]

\[
[t/x]x_0 = x_0 \text{ if } x \neq x_0
\]

\[
[t/x](\text{if } t_1 \text{ then } t_2 \text{ else } t_3) = \begin{cases} \text{if } t/x(t_1) \text{ then } t/x(t_2) & \text{else } t/x(t_3) \\ \text{let } x = t_1 \text{ in } t_2 & \text{let } x = t/x(t_1) \text{ in } t_2 \\ \text{let } x = [t/x][x_1/x_0]t_2 & \text{if } x \neq x_0 \\ x_0 & \text{otherwise} \end{cases}
\]

where \(x_1 = \begin{cases} x_0 & x \notin \text{FV}(t_2) \lor x_0 \notin \text{FV}(t) \\ \min_{x \in \text{VAR}}(x \notin \text{Vars}(t) \cup \text{Vars}(t_2)) & \text{otherwise} \end{cases}\)

Notice that we are using Haskell Curry’s “trick” (described in the appendix of [Barendregt [1985]]) of assuming that VAR has some associated total order (we don’t care about the details of the ordering, just that there is one). This gives us a deterministic mechanism for choosing fresh variables with respect to some relevant terms. This trick is but one of the annoying overly-specific details that we wish to do without. On the other hand, the notion of freshness here is quite precise about which variables the chosen one may not overlap with. This is useful for understanding a system in detail (rather than claiming the need for a “globally fresh variable”, whatever that means, when such a claim is sheer overkill). If we ignore the “minimum” requirement, then we get the truly relevant constraints.

Proposition 2 ([/-/] is well-defined). The equations in Defn. 7 describe a unique function.

\[
F \in \text{CONCRETE}\text{TERM} \times \text{VAR} \times \text{CONCRETE}\text{TERM} \rightarrow \text{CONCRETE}\text{TERM}.
\]

3 Abstract Syntax

Ok, we have concrete syntax, alpha equivalence, and our motivating example of capture-avoiding substitution. Let’s get abstract. First, we exploit a key property of all equivalence relations. An equivalence relation partitions a set into disjoint subsets of equivalent members. We call these subsets equivalence classes. Here’s the definition for our particular case.

Definition 8 (Terms as Alpha-equivalence Classes).

\[
t^a \in \text{TERM} = \text{CONCRETE}\text{TERM}/\sim_a
\]

\[
= \left\{ S \in \mathcal{P} (\text{CONCRETE}\text{TERM}) \mid \begin{cases} \exists t \in \text{CONCRETE}\text{TERM}. t \in S \\ \forall t \in \text{CONCRETE}\text{TERM}. t \in S \iff \forall t_0 \in \text{CONCRETE}\text{TERM}. t_0 \in S \iff t_0 \sim_a t \end{cases} \right\}
\]

This technique of defining an abstract object as an equivalence class has a long tradition going back at least to the mathematician and logician Richard Dedekind. He “defined” the integers \(\mathbb{Z}\) as equivalence classes of pairs of natural numbers, the rationals \(\mathbb{Q}\) as equivalence classes of pairs of integers, and the real numbers \(\mathbb{R}\) as sets of rationals. At each step of construction, he ensured that operations on the “simpler” entity (like addition) could be lifted to the new entities. His goals, and results, were different in some key ways, but the general technique is still the same [Sieg and Schlimm [2005]]

What the...???. And now you can see the punchline laying before you. Our new notion of TERMS is going to be the alpha-equivalence classes of CONCRETETERMS. At first this may seem highly unsatisfying. My first response to it was: “What in the world does it mean for my single program TERM to really stand for a
Definition 9

The formalization of Barendregt’s conventions. To do so we define the them, leaving any necessary injection operations implicit.

The takeaway is: ultimately we will care about the API, not the junk underneath that we use to demonstrate that the API is “implementable”.

The key idea behind Barendregt’s variable convention is that we can encapsulate and abstract away from issues that arise due to bound-variable names by 1) operating over alpha-equivalence classes TERM, but 2) doing so by referencing and working with CONCRETE TERMS as representatives of the underlying alpha-equivalence classes. In short, we can implicitly treat a CONCRETE TERM as whichever alpha-equivalent analogue is most convenient to operate with at that moment. Part 1) corresponds to Convention

set of CONCRETE TERMS? My programs are single trees, not sets of trees! Ohhh that hurts…” Years later, my answer to this is: “Don’t worry: your terms have been strange sets of sets all along, so this is no change at all!” What do I mean by that? Well, remember that set theory at its base is extremely impoverished, just like machine code. Instead of ones and zeros, we have the empty set, the set with the empty set, ad infinitum. We use these entities to encode abstract syntax trees. Going this route ensures that our mathematical entities are rigorously well defined...well, at least as well-defined as set theory, which has held up to lots and lots of “fuzz testing” for the last century, so we’re feeling about as good as we can about it. So the syntax that we are using for abstract syntax trees really represents strange sets, or really families of sets that all satisfy the algebraic properties that we care about. Once we know the relevant algebraic properties, we can ignore the model that satisfies them. That’s how we build up a high-level abstraction. But we want to know that something under the hood could represent that abstraction.

The same thing has happened here: we’ve built up a new family of objects, TERM, in terms of an old abstraction CONCRETE TERM. What we have not yet done is build up our new API, our abstraction, over this new family of objects, such that we can go back to mostly ignoring the low-level details. The weird thing is that our new interface to TERM will look a whole lot like our old interface to CONCRETE TERM, namely constructors and destruction principles. They will look so similar that we will start using the exact same notation for both of them. However, their properties are not exactly the same, and that is where some care is needed to ensure that what you write still makes sense. The analogy is tight, but not perfect.

So yeah, this sort of feels like cheating, but that’s basically the entire dirty secret behind abstraction in computer science: come up with some painfully complicated artifacts, then lovingly wrap them in a human-friendly straight jacket notation for talking about them. Let’s start on building up that interface. For our purposes, it will be very important to map CONCRETE TERMS to their corresponding equivalence classes. This is one of the core tricks underlying the formalization of Barendregt’s conventions. To do so we define the injection function:

Definition 9 (Injection).

\[ A[\cdot] : \text{CONCRETE TERM} \rightarrow \text{TERM} \]
\[ A[t] = \{ t_0 \in \text{CONCRETE TERM} \mid t \sim_a t_0 \} \]

That’s a pretty spartan definition, so let’s draw out some of its implications as theorems. The following properties can be deduced one-by-one, and are somewhat overlapping, but are generally useful, so presented here as a group. They expose some of the properties which I believe that Barendregt intended to exploit in his approach to syntax: operate on TERMS, but mostly by referring to CONCRETE TERMS that represent them, leaving any necessary injection operations implicit.

Proposition 3 (Representation).

1. \( \forall t \in \text{CONCRETE TERM.} \ t \in A[t] \). In words, each representative is an element of the class it represents;
2. \( \forall t_1, t_2 \in \text{CONCRETE TERM.} \ t_1 \sim_a t_2 \iff A[t_1] = A[t_2] \).
   In words, \( A[\cdot] \) exactly conveys the alpha-equivalence classes;
3. \( \forall t^\alpha \in \text{TERM.} \ \forall t \in t^\alpha. \ A[t] = A[t^\alpha] \). In words, any element of \( t^\alpha \) can represent it;
4. \( \forall t^\alpha \in \text{TERM.} \ \exists t \in \text{CONCRETE TERM.} \ t^\alpha = A[t] \). In words, \( A[\cdot] \) is surjective: every TERM can be represented by a CONCRETE TERM.

You’ll see below that we’re going to build on that idea but go further, by developing an algebra for alpha-equivalent syntax, such that everything is precisely specified, leaving behind CONCRETE TERMS as representatives (but we’ll steal their notation).
Following Barendregt’s conventions, the appropriate interpretation is the following:

ERM that arise via injection as those side-conditions hold, each manipulated term can be interpreted as a proposition about the TERMS that arise via injection $\mathcal{A}[\cdot]$.

Here is an example: later we introduce the following reduction rule:

\[
(\lambda x.t) v \rightarrow [v/x]t \quad \text{(sapp)}
\]

However, reduction is defined as operating on TERMS, not CONCRETE TERMS, so this can’t be quite right. Following Barendregt’s conventions, the appropriate interpretation is the following:

\[
\mathcal{A}[(\lambda x.t) v] \rightarrow \mathcal{A}[[v/x]t] \quad \text{(sapp)} \quad BV(\lambda x.t) \cap FV(v) = \emptyset, \quad BV(v) \cap FV(\lambda x.t) = \emptyset
\]

Let me spell this out with even more side-conditions, just to make a bit more explicit what this rule says:

\[
\frac{\exists x \in \text{VAR}, t \in \text{TERM}, v \in \text{VALUE.}}{t_1^x \rightarrow t_2^x} \quad \text{(sapp)} \quad (BV(\lambda x.t) \cap FV(v) = \emptyset) \land (BV(v) \cap FV(\lambda x.t) = \emptyset) \land \\
(t_1^x = \mathcal{A}[(\lambda x.t) v]) \land (t_2^x = \mathcal{A}[[v/x]t]).
\]

Part of why this makes sense is a critical property that requires proof: that capture-avoiding substitution preserves alpha-equivalence.

So this inductive rule is really about relating TERMS to one another, but that relationship is expressed in terms of CONCRETE TERMS and the relationships that hold between them. Two TERMS are related by reduction only if constraints on two CONCRETE TERMS, which inject to them, can be satisfied. So since the description is primarily in terms of CONCRETE TERMS, the variable convention suggests that we leave implicit the fact that the description is ultimately about TERMS, and phrase the constraints as a universal assumption that need never be repeated.

However, when working on paper it is easy to accidentally cross abstraction boundaries, or not precisely understand where the boundaries lie (because there is no notation to help you “type check” your formalism: you have to do “injection-inference” in your head, which is easy to get wrong. This technique of quotienting carries over to other language aspects. For example, in a language with memory addresses, how do we formally express that we don’t care which memory address is mapped to a value? Felleisen and Friedman [1989] demonstrate this in action.

One key principle behind this quotienting is that the most relevant reasoning principles for CONCRETE TERM can be lifted to analogous principles for TERM, at first through the lens of $\mathcal{A}[^{\cdot}]$, but then we devise a high-level API for terms by introducing TERM “constructors”, functions that abstract away the gory details of equivalence classes.

First, consider some quite-standard backward-reasoning lemmas for CONCRETE TERM

**Lemma 1** (Inversion). Let $t \in \text{CONCRETE TERM}$. Then exactly one of the following is true:

1. $\exists n \in \mathbb{Z}. \ t = n$;
2. $\exists t_1, t_2 \in \text{CONCRETE TERM}. \ t = t_1 + t_2$;
3. $\exists x \in \text{VAR}. \ t = x$;
4. $\exists x \in \text{VAR}. \ t_0 \in \text{CONCRETE TERM}. \ t = \lambda x. t_0$;
5. $\exists t_1, t_2 \in \text{CONCRETE TERM}. \ t = t_1 t_2$;
6. $\exists x \in \text{VAR}. \ t_1, t_2 \in \text{CONCRETE TERM}. \ t = \text{let } x = t_1 \text{ in } t_2$.

Using $\mathcal{A}[^{\cdot}]$, we can state and prove the following reasoning lemmas for TERM, which quite intentionally look like the ones for CONCRETE TERM, but for that pesky $\mathcal{A}[^{\cdot}]$.

**Lemma 2** (Representational Inversion). Let $t^\alpha \in \text{TERM}$. Then exactly one of the following is true:
1. \( \exists n \in \mathbb{Z}. \, t^n = A[n] \);
2. \( \exists t_1, t_2 \in \text{ConcreteTerm}. \, t^n = A[t_1 + t_2] \);
3. \( \exists x \in \text{VAR}. \, t^n = A[x] \);
4. \( \exists x \in \text{VAR}, t \in \text{ConcreteTerm}. \, t^n = A[\lambda x.t] \);
5. \( \exists t_1, t_2 \in \text{ConcreteTerm}. \, t^n = A[t_1 t_2] \);
6. \( \exists x \in \text{VAR}, t_1, t_2 \in \text{ConcreteTerm}. \, t^n = A[\text{let } x = t_1 \text{ in } t_2] \).

Notice that some of the new propositions no longer assert unique existence; we’ll talk about that shortly.

4 Beyond Barendregt: Alpha-equivalent Syntax

**TERM “constructors”**  Now we deal with the pesky \( A[\ ] \), or by hiding it behind some more abstracting machinery. In particular we lift \text{ConcreteTerm} constructors to \text{TERM} “constructor” operations.

The definitions of the operators here are “elementary” in the sense that they are simple definitions (easily seen to be well-defined), though these equations do not adequately express the fundamental principles of interest: we state these properties immediately afterward. Technically we could state the fundamental equations and then prove that they definitely describe a function instead. It’s a little weird since ultimately we want to use a “nondeterministic” representation description that uses a \text{ConcreteTerm} \( t \) to denote a \text{TERM} \( t^n \).

**Definition 10 (TERM Operators).**
\[
\begin{align*}
\cdot^n & : (\mathbb{Z} \cup \text{VAR}) \to \text{TERM} \\
n^n & = A[n] \\
x^n & = A[x] \\
\odot & : \text{TERM} \times \text{TERM} \to \text{TERM} \\
t_1^n \odot t_2^n & = \{ (t_1, t_2) \in \text{TERM} \mid t_1 \in t_1^n, t_2 \in t_2^n \} \\
+ & : \text{TERM} \times \text{TERM} \to \text{TERM} \\
t_1^n + t_2^n & = \{ t_1 + t_2 \in \text{TERM} \mid t_1 \in t_1^n, t_2 \in t_2^n \} \\
\lambda^n & : \text{VAR} \to \text{TERM} \to \text{TERM} \\
\lambda^n[x]t^n & = \{ \lambda x.t \in \text{TERM} \mid t \in t^n \} \\
\text{let}^n & : \text{VAR} \times \text{TERM} \times \text{TERM} \to \text{TERM} \\
\text{let}^n(x, t_1^n, t_2^n) & = \{ \text{let } x = t_1 \text{ in } t_2 \in \text{TERM} \mid t_1 \in t_1^n, t_2 \in t_2^n \}
\end{align*}
\]

The definitions above are “elementary”, requiring only simple reasoning to justify the existence of each function. The equational reasoning principles that we really intend to work with follow:

**Proposition 4 (Liftings).**
1. \( \forall t_1, t_2 \in \text{ConcreteTerm}. \, A[t_1] \odot A[t_2] = A[t_1 t_2] \);
2. \( \forall t_1, t_2 \in \text{ConcreteTerm}. \, A[t_1] + A[t_2] = A[t_1 + t_2] \);
3. \( \forall x \in \text{VAR}, t \in \text{TERM}. \, A^n[x, A[t]] = A[\lambda x.t] \).

In principle we could have defined these three operators directly as the unique functions that satisfy the equations stated in the proposition, but proving that these propositions (which on the surface are about \text{ConcreteTerms}) justify definitions of the given functions (which ultimately are about \text{Terms}) require additional reasoning beyond what we usually take for granted in an equational function definition. So we separate concerns: define the functions in an elementary way, and then separately build the reasoning principles that we will use when working with these functions.
Finally, take note of the atomic injection operation \( ^\alpha \), which embeds numbers and variables into our \( \text{TERM} \) classes. This concept makes direct sense in \( \text{CONCRETE} \), where a variable reference \( x \) is not the same thing as a binder \( \lambda x \). You can think of the variable reference as being \( \text{var}[x] \), an operator indexed on a particular variable, and a number being \( \text{num}[n] \), an operator indexed on a particular number. So \( \mathbb{N} \) and \( \text{VAR} \) are injected into \( \text{CONCRETE} \), even though our notation makes that implicit. Our approach to \( \text{TERM} \) makes it explicit.

Now armed with operators that lift our term construction principles from \( \text{CONCRETE} \) to \( \text{TERM} \), we can analogously lift our \( \text{TERM} \) analysis principles:

**Proposition 5 (Structural Inversion).** Let \( t^\alpha \in \text{TERM} \). Then exactly one of the following is true:

1. \( \exists n \in \mathbb{Z}. t^\alpha = n^\alpha \);
2. \( \exists \alpha_1, \alpha_2 \in \text{TERM}. t^\alpha = t_1^\alpha +^\alpha t_2^\alpha \);
3. \( \exists x \in \text{VAR}. t^\alpha = x^\alpha \);
4. \( \exists \alpha \in \text{VAR}. t^\alpha \in \text{TERM}. t^\alpha = \lambda^\alpha[x]t^\alpha \);
5. \( \exists \alpha_1, \alpha_2 \in \text{TERM}. t^\alpha = t_1^\alpha \otimes^\alpha t_2^\alpha \);
6. \( \exists \alpha \in \text{VAR}. t_1^\alpha, t_2^\alpha \in \text{TERM}. t^\alpha = \text{let}^\alpha(x, t_1^\alpha, t_2^\alpha). \)

One key difference between structural inversion on alpha-equivalence classes versus concrete terms is that inversion on lambda abstractions and let-bindings do not decompose \( \text{TERM} \) into unique constituents. A \( \text{TERM} \) \( \text{AR}[\lambda x. t] \) decomposes into *many* possible \( \text{VAR} \)s \( x \) and \( \text{TERM} \)s \( t^\alpha \). This loss of uniqueness is both a boon and a curse. We can prove that all but a *finite* number of variables \( x \) can arise, or in a sense be demanded. We’ll see that that finite number of unavailable variables corresponds to the proper notion of free variables for \( \text{TERM} \), and is sometimes called its *finite support* (a term from nominal logic, a theory for modeling binders developed by Andrew Pitts and colleagues [Pitts 2013]). This leads to extremely useful, more powerful principles for terms with bound variables.

**Proposition 6 (Structural Inversion with Finite Support).** Let \( t^\alpha \in \text{TERM} \). Then

1. \( \forall t^\alpha \in \text{TERM}. (\exists x_1 \in \text{VAR}. t_1^\alpha \in \text{TERM}. t^\alpha = \lambda^\alpha[x_1]t_1^\alpha) \implies \forall \alpha \in \mathcal{P}^{\text{fin}}(\text{VAR}). \exists x_2 \in \text{VAR}. t_2^\alpha \in \text{TERM}. x_2 \notin \alpha \wedge t^\alpha = \lambda^\alpha[x_2]t_2^\alpha. \)
2. \( \forall t^\alpha \in \text{TERM}. (\exists x_1 \in \text{VAR}. t_{11}^\alpha, t_{12}^\alpha \in \text{TERM}. t^\alpha = \text{let}^\alpha(x_1, t_{11}^\alpha, t_{12}^\alpha) \implies \forall \alpha \in \mathcal{P}^{\text{fin}}(\text{VAR}). \exists x_2 \in \text{VAR}. t_{21}^\alpha, t_{22}^\alpha \in \text{TERM}. x_2 \notin \alpha \wedge t^\alpha = \text{let}^\alpha(x_2, t_{21}^\alpha, t_{22}^\alpha). \)

In essence, these propositions guarantee that it is always possible to decompose a term with a binder such that the newly free variable is surely not in some freely chooseable finite set of variables \( \alpha \).

It may be a little confusing that the notation \( t^\alpha \) is used for a whole metavariable, whereas \( n^\alpha \) and \( x^\alpha \) are applications of a function \( ^\alpha \) to numbers \( n \) and \( \text{VARs} \) \( x \) respectively. I’ve allowed myself to use a bit of *notational punning*, so as to avoid inventing distinct notation for \( \text{TERM} \) metavariables, while still alluding to this concept lifting into alpha-equivalence classes. I hope this bit of syntactic sugar is not too confusing.

### 4.1 Induction On Alpha-Equivalence Classes

The direct definition of \( \text{TERM} \) does not automatically induce the kind of reasoning principle that we are used to exploiting for syntax. Here we develop a bespoke induction principle for \( \text{TERM} \), in painful detail. First, following the usual strategy, we define “some structural term subset” \( \text{ST} \) of \( \text{TERM} \) (hint, it’s not just *any* subset).

**Definition 11 (Structural Term).** \( \text{ST} \subseteq \text{TERM} \)

\[
\begin{align*}
\frac{n^\alpha \in \text{ST}}{(\text{STn})} & \quad \frac{t_1^\alpha \in \text{ST} \quad t_2^\alpha \in \text{ST}}{(\text{ST+})} & \quad \frac{x^\alpha \in \text{ST}}{(\text{STx})} & \quad \frac{t^\alpha \in \text{ST}}{(\text{ST\lambda})} \\
\frac{t_1^\alpha \in \text{ST} \quad t_2^\alpha \in \text{ST}}{(\text{ST@})} & \quad \frac{\text{let}^\alpha(x, t_1^\alpha, t_2^\alpha) \in \text{ST}}{(\text{STlet})}
\end{align*}
\]
Then we conveniently discover that the subset we’ve identified inductively is in fact the entire set, thereby letting us use the new induction principle to reason about the old set.

**Proposition 7** (Redefinition). \( \text{TERM} \subseteq \text{ST} \).

**Corollary 1.** \( \text{TERM} = \text{ST} \).

And voila! By extensionality \( \text{TERM} \) inherits \( \text{ST} \)'s induction principle:

**Proposition 8** (Principle of Rule Induction on the structure of \( \text{TERM} \)). Let \( \Phi \) be a predicate on terms \( t^\alpha \). Then \( \Phi(t^\alpha) \) holds for all \( t^\alpha \in \text{TERM} \) if:

1. \( \forall n \in \mathbb{N}. \Phi(n^\alpha) \);
2. \( \forall t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha +^\alpha t_2^\alpha) \);
3. \( \forall x \in \text{VAR}. \Phi(x^\alpha) \);
4. \( \forall x \in \text{VAR}, t^\alpha \in \text{TERM}. \Phi(t^\alpha) \implies \Phi(\lambda x^\alpha[x]t^\alpha) \);
5. \( \forall t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha \@^\alpha t_2^\alpha) \);
6. \( \forall x \in \text{VAR}, t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(\text{let}^\alpha(x, t_1^\alpha, t_2^\alpha)) \).

The above inductive definition and corresponding induction principle are appealing: they make \( \text{TERM} \) look and feel a lot like \( \text{CONCRETE TERM} \), providing analogous reasoning principles. However, we sometimes want stronger reasoning principles for \( \text{TERM} \), which give us even stronger “induction hypotheses” when performing proof by induction.

To explain, consider the induction lemma induced for \( \Lambda \), Prop. 8.4. It says that having assumed a particular variable \( x \) and a particular \( t^\alpha \) and the property \( \Phi \) holding for that particular \( t^\alpha \), we must prove that

\[
\Phi(\lambda x^\alpha[x](t^\alpha)) \text{ holds. The unsatisfying thing about this lemma is that a term }\lambda x^\alpha[x](t^\alpha)\text{ can be represented by many different } x', t^\alpha' \text{ pairs, such that }\lambda x^\alpha[x'](t^\alpha') = \lambda x^\alpha[x](t^\alpha), \text{ and at different points in a proof we might benefit from considering other such decompositions. Unfortunately, the above principle does not let us assume that }\Phi(t^\alpha'),\text{ only }\Phi(t^\alpha).\text{ This can be a severe limitation, especially when combining induction with backward reasoning. Luckily we can do better.}
\]

First, we introduce a new well-founded relation on \( \text{TERM} \) that is motivated by the structure that we have superimposed on the set.

\[
\alpha\text{-Structural Order}
\]

\[
t_1^\alpha \sqsubseteq t_1^\alpha \@^\alpha t_2^\alpha \quad t_2^\alpha \sqsupseteq t_1^\alpha \@^\alpha t_2^\alpha \quad t^\alpha \sqsubseteq \lambda^\alpha[x](\hat{t}) \quad t_1^\alpha \sqsubseteq \text{let}^\alpha(x, t_1^\alpha, t_2^\alpha) \quad t_2^\alpha \sqsubseteq \text{let}^\alpha(x, t_1^\alpha, t_2^\alpha)
\]

To prove well-foundedness, it suffices to show that \( t_1^\alpha \sqsubseteq t_2^\alpha \) implies \( \text{size}(t_1^\alpha) < \text{size}(t_2^\alpha) \) given an appropriate size function for \( \text{TERM} \). We discuss how to use the size function for \( \text{CONCRETE TERM} \) to define the analogous function for \( \text{TERM} \) in the next section.

The \( \sqsubseteq \) relation induces a principle of well-founded induction on \( \text{TERM} \) that exhibits the power that we seek. The key observation is that the standard rule for well-founded induction collects all elements \( t_1^\alpha \) such that \( t_2^\alpha \sqsubseteq t_1^\alpha \) as the premises for the rule with conclusion \( t_2^\alpha \). This means that the instances of the rule for a lambda abstraction will have the form:

\[
\frac{\{ t^\alpha \in \text{TERM} \mid \exists x \in \text{VAR}. \lambda x^\alpha[x](t^\alpha) = \lambda x^\alpha[x_0]t_0^\alpha \}}{\lambda^\alpha[x_0]t_0^\alpha} \quad (\sqsubseteq \lambda)
\]

Thus, the premise includes all candidate \( \text{TERM} \) bodies. We can then take the corresponding induction principle and explicitly distinguish the structure of \( \text{TERM} \) to yield the following more expressive one.
Proposition 9 (Principle of Rule Induction on the $\alpha$–structure of TERM). Let $\Phi$ be a predicate on TERMS $t^\alpha$. Then $\Phi(t^\alpha)$ holds for all $t^\alpha \in$ TERM if:

1. $\forall n \in \mathbb{N}. \Phi(n^\alpha)$;
2. $\forall t_1^\alpha, t_2^\alpha \in$ TERM. $\Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha +^\alpha t_2^\alpha)$;
3. $\forall x \in$ VAR. $\Phi(x^\alpha)$;
4. $\forall x \in$ VAR, $t^\alpha \in$ TERM. $(\forall x_0 \in$ VAR, $t_0^\alpha \in$ TERM. $\lambda^\alpha[x_0]t^\alpha_0 = \lambda^\alpha[x]t^\alpha \implies \Phi(t_0^\alpha)) \implies \Phi(\lambda^\alpha[x]t^\alpha)$;
5. $\forall t_1^\alpha, t_2^\alpha \in$ TERM. $\Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha @^\alpha t_2^\alpha)$;
6. $\forall x \in$ VAR, $t_1^\alpha, t_2^\alpha \in$ TERM. $\Phi(t_1^\alpha) \land (\forall x_0 \in$ VAR, $t_0^\alpha \in$ TERM. let$^\alpha(x_0, t_1^\alpha, t_0^\alpha) = let^\alpha(x, t_1^\alpha, t_2^\alpha) \implies \Phi(t_0^\alpha)) \implies \Phi(let^\alpha(x, t_1^\alpha, t_2^\alpha))$.

It may seem odd that we are not keeping the corresponding binding variables $x_0$ around, but since our reasoning says that there must exist such an $x_0$, we just "pick/choose" such an $x_0$ when we need it.

One last observation: it is often the case that a proof using $\alpha$-structural Induction could also have been performed using Induction on the size of TERM, and the proof would have a similar structure. Can you see why?

4.2 Lifting Functions

One of the keys to our smooth development of abstract syntax is that we can adapt many functions from concrete syntax with minimal effort. In particular, we have the following proposition

Proposition 10 (Lifting). Let $S$ be some set, and $F : \text{CONCRETETERM} \to S$. Then if $\forall t_1, t_2 \in \text{CONCRETETERM}. t_1 \sim_\alpha t_2 \implies F(t_1) = F(t_2)$, then there is a unique function $F^\alpha : \text{TERM} \to S$ such that $\forall t \in \text{CONCRETETERM}. F^\alpha(A[t]) = F(t)$.

Thus it suffices to prove that a function is invariant for alpha-equivalent terms in order to justify a lifted version of a function. Here is an example.

Proposition 11. $\forall t_1, t_2 \in \text{CONCRETETERM}. t_1 \sim_\alpha t_2 \implies FV(t_1) = FV(t_2)$.

This proposition gives us a natural (expected) notion of free variables for TERMS. We can recast it equationally by proving the analogue of each equation post-hoc, e.g.:

Proposition 12. $\forall x \in$ VAR, $t^\alpha \in$ TERM. $FV^\alpha(\lambda^\alpha[x]t^\alpha) = FV^\alpha(t^\alpha) \setminus \{x\}$.

On the other hand, notions like all occurring variables Vars or especially bound variables BV, have no immediately direct analogue in TERM. The latter fact is in essence the entire point of our construction: bound variable names are insignificant. Here is a typical definition of a bound variables function:

Definition 12 (Bound Variables).

$$BV : \text{CONCRETETERM} \to \mathcal{P}(\text{VAR})$$

$$BV(n) = \emptyset$$

$$BV(t_1 + t_2) = BV(t_1) \cup BV(t_2)$$

$$BV(x) = \emptyset$$

$$BV(\lambda x.t) = BV(t) \cup \{x\}$$

$$BV(t_1 t_2) = BV(t_1) \cup BV(t_2)$$

$$BV(let x = t_1 in t_2) = BV(t_1) \cup (BV(t_2) \cup \{x\})$$

Note that $\lambda y.y \sim_\alpha \lambda x.x$ but $BV(\lambda y.y) = \{y\} \neq \{x\} = BV(\lambda x.x)$. In fact, $A[\lambda x.x] = \{\lambda x.x \mid x \in \text{VAR}\}$, so in a sense every variable whatsoever appears bound in this alpha-equivalence class! Thus you could argue that the concept of bound variables has little to no meaning here. Indeed!
4.3 Lifting Capture-Avoiding Substitution

Just as we were able to use concrete term construction and concrete term analysis to define analogues for our abstract terms, so can we now do so for many other operations on abstract syntax. **BUT NOT ALL:** some notions don’t make sense on alpha-equivalence classes. For example, Naive substitution does not have an obvious analogue for `TERM`: in fact that was kind of the point. However, it is important that if we wish `TERM` to be our idealized syntax, then we must make sure that the operations we want CAN be lifted to `TERM`.

[RG: Demonstrate how/where naive substitution breaks down, and where it succeeds].

However, capture-avoiding substitution does remain meaningful after we lift our syntax from `CONCRETETERM` to `TERM`!

**Proposition 13** (Capture-Avoiding Substitution Preserves Alpha-equivalence).

\[ \forall x \in \text{VAR}, t_{11}, t_{12}, t_{21}, t_{22} \in \text{CONCRETETERM}. t_{11} \sim_a t_{21} \land t_{12} \sim_a t_{22} \implies [t_{11}/x]t_{12} \sim_a [t_{21}/x]t_{22}. \]

The above proposition differs from our earlier ones in a couple of interesting ways. First, capture-avoiding substitution is a function of **three arguments:** two `CONCRETETERM`s and one `VAR`. Technically, capture-avoiding substitution preserves alpha equivalence in several ways:

1. two `CONCRETETERM`s being substituted into the same target `CONCRETETERM` may be alpha-equivalent;
2. two `CONCRETETERM`s being targeted with substitution by the same `CONCRETETERM` may be alpha-equivalent;
3. two alpha-equivalent `CONCRETETERM`s may be substituted respectively into two other alpha-equivalent `CONCRETETERM`s.

Since any `CONCRETETERM` is alpha-equivalent to itself, the third case subsumes the first two.

Another interesting aspect of this situation is the outcome: In the case of free variables or size, the results for equivalent inputs were equal. In the case of substitution, the outputs, which are `CONCRETETERM`s need only be **alpha-equivalent**. So our lifting of capture-avoiding substitution does not yield a `CONCRETETERM`, but rather a `TERM`, just like our lifted constructors. That’s easy to address: compose capture avoiding substitution with lifting and you get the desirable property:

**Corollary 2** (Capture-Avoiding Substitution Lifts). Let `F : CONCRETETERM \times \text{VAR} \times \text{CONCRETETERM} \to \text{TERM}` be defined by `F = A[\cdot] \circ [\cdot/\cdot]`. Then

\[ \forall x \in \text{VAR}, t_{11}, t_{12}, t_{21}, t_{22} \in \text{CONCRETETERM}. t_{11} \sim_a t_{21} \land t_{12} \sim_a t_{22} \implies F(t_{11}, x, t_{12}) = F(t_{21}, x, t_{22}). \]

Now we can apply Prop 10 to \((A[\cdot] \circ [\cdot/\cdot])\) to lift capture-avoiding substitution.

**Definition 13** (Lifted Substitution).

\[ [\cdot/\cdot]^\alpha : \text{TERM} \times \text{VAR} \times \text{TERM} \to \text{TERM} \]

\[ [A[t_1]/x]^\alpha A[t_2] = A[[t_1/x]^\alpha t_2]. \]

Ultimately with some work [RG: Do it], we can arrive at an equational description of lifted substitution that looks a lot like our earlier one for naive substitution.

**Proposition 14** ((Capture-avoiding) Substitution, Recursive-ishly). The `[\cdot/\cdot]^\alpha` function is the function in `TERM \times \text{VAR} \times \text{TERM}` uniquely described by the following equations:

\[ [t^n/x]^\alpha n = n \]
\[ [t^n/x]^\alpha(t^n_1 +^\alpha t^n_2) = ([t^n/x]^\alpha t^n_1) +^\alpha ([t^n/x]^\alpha t^n_2) \]
\[ [t^n/x]^\alpha x^\alpha = t^n \]
\[ [t^n/x]^\alpha x_0^\alpha = x_0^\alpha \text{ if } x_0 \neq x \]
\[ [t^n/x]^\alpha let(x_0, t^n_1, t^n_2) = let^\alpha(x_0, ([t^n/x]^\alpha t^n_1), ([t^n/x]^\alpha t^n_2)) \text{ if } x \neq x_0 \]
\[ [t^n/x]^\alpha (\lambda x^\alpha [x_0]t^n_0) = \lambda x^\alpha [x_0]([t^n/x]^\alpha t^n_0) \text{ if } x \neq x_0 \]
One might want to rewrite this definition using “top-level injection”, but that would lead to some “type errors”, in the sense that you cannot apply the lifted substitution to individual terms...you’d end up referring to substitution on individual terms, which is where we started.

4.4 Pay no attention to the sets behind the curtain!

“The relations or laws which are derived entirely from the conditions …and therefore are always the same in all ordered simply infinite systems, whatever names may happen to be given to the individual elements, form the next object of the science of numbers or arithmetic.” – Richard Dedekind (as translated in Sieg and Schlimm [2005])

Now we can put the final seal on our new abstraction layer, and pay no further mind to the messy details. The following may feel like some notational sleight of hand, but it’s so much more than that: it’s just another example of building the higher-level tools and abstractions that you need and want.

In the above development, we have defined a new notion of TERMS as somewhat unpleasant nebulous blobs of sets of CONCRETE TERMS, our old notion, by taking the concept of alpha-equivalence to turn mere equivalence relationships into set-theoretic identities. Then, we tamed these bloppy sets by building up a more pleasant “structural” interface to them: we constructed operators like \( \lambda x \) and \( \otimes \), which are directly analogous to our constructors for CONCRETE TERMS. We also lifted capture-avoiding substitution to form a corresponding notion of substitution, which is by its very nature capture-avoiding: our old naïve substitution doesn’t even make sense for TERM. And finally we developed inversion principles, and “structural” induction principles.

With this nice clean syntax-like API for TERM, we don’t really care what a TERM is made up of internally: we can ignore the internals and just use our construction, destruction, and induction principles just like CONCRETE TERMS. So that’s exactly what we’re going to do: from now on we change notation, using the original notation for CONCRETE TERMS to refer to our constructed operators for TERMS! We don’t care about CONCRETE TERMS anymore, so our notation need never refer to them again. This is kind of like shadowing native language operator names with your own (as we can do with ease in a language like Racket!).

Definition 14 (Abstract Syntax).

\[
\begin{align*}
& x \in \text{VAR}, \ n \in \mathbb{Z}, \ t \in \text{TERM} \\
& t \ ::= \ n \mid t + t \mid x \mid \lambda x. t \mid t t \mid \text{let } x = t \text{ in } t
\end{align*}
\]

Note: TERMS are identified up-to choice of bound variable names.

What you may not realize or remember is that we use an exactly analogous technique to give meaning to CONCRETE TERM in the first place! There is no such thing as an “abstract syntax tree” in set theory: we build up the concept of the set TREE and then inductively define CONCRETE TERM. Then we show that we never really cared about a concrete notion of TREE, but rather just the algebraic structure induced by the constructors: any model of those constructors will do. This is what Dedekind alludes to in the quote that begins this subsection: whether one uses Roman numerals, Coq-like unary numerals, binary numerals, or Indo-Arabic numerals, all of these notations represent a singular object of science: the natural numbers. We seek a pleasant notation and reasoning principles for “terms up to choice of bound variable names.”

Here we’ve done the same exact thing, but the resulting algebraic structure, of “terms modulo choice of bound variable name” is more involved of a construction. But ideally our construction process has given you confidence that we have defined the right kind of structure, and deduced the right structural properties in terms of our abstract constructor operations. Now we can treat TERM as an opaque set of constructors with interesting identities.

Def. [14] is a phenomenal compression of the entire construction process that we described above, corresponding in essence to Def. [10] which gives construction principles, Prop. [5] which gives destruction principles, and Prop. [8] which gives induction principles, which is just as we typically interpret BNFs, though for a much more straightforward notion of syntax.

In essence, this is what Barendregt alludes to as a practice, albeit with fewer of the details spelled out (like the reasoning principles), and some anachronisms sneaking in from CONCRETE TERM (e.g., a TERM has
no coherent notion of “bound variables”, though it does have a coherent notion of “free variables” which affects how formulae can be precisely interpreted.

Some caution must still be exercised, though, when we consider the definition of new relations and functions over \( \text{TTERM} \), thanks to the fact that it is not a simple inductive definition, and its inductive characterisation and corresponding principle of induction is not the same as your typical syntactic definition. In particular, Defn. \([11]\) is not a deterministic inductive definition: because of choice of bound variables, a member of \( \text{ST} \) can be constructed multiple ways. This means that we do not as straightforwardly get a corresponding Principle of Definition by Recursion: we must ensure that the given equations are invariant with respect to choice of bound variables.

For example, here is a legitimate (re)-definition of capture-avoiding now with our new notation.

**Definition 15** (Substitution (with abstract \( \text{TTERM} \)).

\[
\frac{[t/x]n = n}{[t/x](t_1 + t_2) = ([t/x]t_1) + ([t/x]t_2)}
\]

\[
[t/x]x = t
\]

\[
[t/x]x_0 = x_0 \quad \text{if} \quad x_0 \neq x
\]

\[
[t/x](t_1 t_2) = ([t/x]t_1) ([t/x]t_2)
\]

\[
[t/x]let \quad x_0 = t_1 \quad \text{in} \quad t_2 = let \quad x_0 = ([t/x]t_1) \quad \text{in} \quad ([t/x]t_2) \quad \text{if} \quad x \neq x_0
\]

\[
[t/x]\lambda x_0. t_0 = \lambda x_0.[t/x]t_0 \quad \text{if} \quad x \neq x_0
\]

This definition is meant to be interpreted exactly the same as Defn. \(??: \) only the notation is different. Recall though that this function definition was justified by a process of lifting Defn. \(??\) from \( \text{CONCRETE\text{TERM}} \) to \( \text{TERM} \).

In particular, if we were to compare this to the Principle of Definition by Recursion for \( \text{CONCRETE\text{TERM}} \), we would see that this does not match that recursion schema. To see this, it’s worth comparing the above definition to Def. \(??\), the one for Naïve substitution. Aside from the particulars of the substitution notation, each equation looks like one from Naïve substitution, but there are two fewer equations here: the equations for lambda and let from naïve substitution that handle collisions with matching bound variables have been omitted. Technically such equations could be added as well, but they would be redundant here. In the case of Naïve substitution, they are needed to complete the definite description. Here they are just extra properties.

Must we always define functions over \( \text{TERM} \) by appealing back to \( \text{CONCRETE\text{TERM}} \) and lifting the result every time??? That would really suck: it would mean that our new abstraction does not enjoy all of the benefits that we have come to enjoy from BNFs and other inductive definition abstractions. In short, we need a definition principle that could rightfully be called \( \alpha \)-structural recursion for \( \text{TERM} \). Such a principle should allow us to state the definition of functions over \( \text{TERM} \) as equations that we know to definitely describe a unique function merely by consulting the shape (i.e., schema) of the relevant equations.

[RG: Flesh this idea out here]

## 5 Semantics

Okay, now we have a rigorous syntax for our language, let’s write some semantics. The funny thing is that because of the notational sleight-of-hand, the results look the same as they did without alpha-equivalence classes. But to clarify what is going on, we’ll also desugar some of the definitions just so you can see where exactly all this machinery that we’ve developed (and now proceed to hide aggressively) comes into play.
Definition 16 (Dynamic Semantics Artifacts).

\[ v \in \text{VALUE} \subseteq \text{PGM}, \quad F \in \text{FRAME}, \quad r \in \text{REDEX} \subseteq \text{PGM} \]
\[ f \in \text{FAULTY} \subseteq \text{REDEX}, \quad \text{err} \in \text{ERROR}, \quad c \in \text{CONFIG} \]
\[ p \in \text{PGM} = \{ t \in \text{TERM} \mid FV(t) = \emptyset \}, \quad \text{obs} = \{ \text{procedure}, \infty \} \cup \mathbb{Z} \cup \text{ERROR} \]

\[ v ::= n \mid \lambda x.t \]
\[ F ::= \square p \mid v \mid \text{let } x = \square \text{ in } t \mid \square + p \mid v + \square \]
\[ r ::= v v \mid \text{let } x = v \text{ in } t \mid v + v \]
\[ f ::= v_1 v_2 \quad v_1 \notin \{ \lambda x.t \in \text{TERM} \} \]
\[ \mid v_1 + v_2 \quad \{ v_1, v_2 \} \notin \mathbb{Z} \]
\[ \text{err} ::= \text{mismatch} \]
\[ c ::= p \mid \text{err} \]

We’ll begin our unpacking with the definition of PGM, the set of programs. Don’t let the metavariable \( t \) fool you, we really mean \( t^\alpha \). And we’re not using the free variable function on CONCRETE TERM either. We’ve just dropped the \(^\alpha\). So really:

\[ \text{PGM} = \{ t^\alpha \in \text{TERM} \mid FV^\alpha(t^\alpha) = \emptyset \} \]

Ok, not bad so far.

Now for the definition of VALUE, which is a subset of PGM, we’re using the subtle trick that values must be closed, so the BNF can only be interpreted as a shorthand for a trivial inductive definition of a subset of PGM:

\[ \text{VALUE} \subseteq \text{PGM} \]

\[ n^\alpha \in \text{VALUE} \]
\[ \lambda^\alpha[x]t^\alpha \in \text{VALUE} \]
\[ t^\alpha \in \text{TERM} \]

The second rule is the interesting one. Since VALUE is defined as a subset of PGM, we automatically get the constraint that \( \lambda^\alpha[x]t^\alpha \in \text{PGM} \) for any legal instance of this rule. Note as well that the rule has no premise, just a side-condition. So this is really just definition by cases.

The definition of FRAME is rife with syntactic sugar as always. There are no special uses of any \text{TERM} or CONCRETE TERM constructors anywhere. The syntax we use, as usual, alludes to the role of each kind of frame and how it behaves when plugged with a PGM.

The definition of FAULTY is most precisely presented using something other than faux BNF, as done here: we have abused BNF quite badly. But BNF is such a common social standard for syntax definition, so it is hard to stray from it. Like VALUE, this too can be interpreted as trivial inductive definitions but with more interesting side-conditions.

Ok, now we can consider the structural operational semantics for the language.

Definition 17 (Single-step Reduction).

\[ \lambda x.t \xrightarrow{\text{sapp}} [v/x]t \quad (\text{sapp}) \]
\[ \text{let } x = v \text{ in } t \xrightarrow{\text{slet}} [v/x]t \quad (\text{slet}) \]
\[ p_1 \xrightarrow{\text{Fstep}} p_2 \quad \text{F}(p_1) \rightarrow F[p_2] \quad (\text{Fstep}) \]
\[ p_1 \xrightarrow{\text{err}} \quad \text{F}(p_1) \rightarrow \text{err} \quad (\text{Ferr}) \]

Let’s take one example and work through it. The (sapp) rule can be precisely rendered as follows:

\[ (\lambda^\alpha[x]t^\alpha) @^\alpha v \xrightarrow{\text{sapp}} [v/x]^\alpha t^\alpha \]

Notice the use of \text{TERM} “constructors” to structure this rule. Here is where our representation is very much non-algorithmic because of the use of higher-level operators. But this is most in-line with what is meant throughout. Conveniently enough we could rewrite this rule using properties of all of these operators, to phrase it all in terms of injection:

\[ \mathcal{A}((\lambda x.t_1) t_2) \rightarrow \mathcal{A}(t_2/x)t_2 \quad (\text{sapp}) \]

\( t_2 \in v \)
This rephrasing of the rule says exactly the same thing, albeit with a bit less abstraction. Notice that in order to precisely phrase it, I had to grab some representative out of \( v \), which is some alpha-equivalence class of \( \text{CONCERETERM} \). This is arguably a violation of abstraction boundaries, but then using \( A[\cdot] \) can also be considered such a violation.

One nice thing about this latter presentation is its implications for language implementors. The language is defined over equivalence classes, but the intrepid programmer may represent those classes using any representative that works. This means that any technique for freshening variables (where needed) is considered valid, since the representative is just a stand-in for its entire class (via \( A[\cdot] \)). For example, when performing substitution, no longer are we tied to Curry’s particular choice of fresh variable names, which is fantastically annoying to implement. Any fresh name generation mechanism is fair game.

Divergence is pretty abstract, referring to single-step reduction as a side-condition, so there’s nothing to worry about here.

### Definition 18 (Divergence).

\[
\begin{array}{c}
p_1 \rightarrow p_2 \quad p_2 \xrightarrow{\infty} \\
\hline
p_1 \xrightarrow{\infty}
\end{array}
\]

Evaluation benefits from some treatment.

### Definition 19 (Evaluation).

\[
eval : \text{PGM} \rightarrow \text{OBS}
\]

\[
\begin{align*}
eval(p) &= n & \text{if } p \rightarrow^* n \\
eval(p) &= \text{procedure} & \text{if } p \rightarrow^* \lambda x.t \\
eval(p) &= \text{err} & \text{if } p \rightarrow^* \text{err} \\
eval(p) &= \infty & \text{if } p \xrightarrow{\infty}
\end{align*}
\]

Since \( \text{OBS} \) includes literal numbers, rather than (boring) alpha-equivalence classes of numbers, we need to unpack a bit. Similarly, procedures should be desugared. The rest is as-is.

\[
\begin{align*}
eval(p) &= n & \text{if } p \rightarrow^* n^\alpha \\
eval(p) &= \text{procedure} & \text{if } p \rightarrow^* \lambda^\alpha[x]t^\alpha
\end{align*}
\]

### 5.1 Redux

Having shown how some of the constructs desugar above, I hope that it is clearer that there are some subtleties involved. The \((\text{sapp})\) rule only makes sense because \( [\cdot]^{\alpha} \) is well-defined, which requires proof. Furthermore, there is some work involved to insert “constructors” or injections in all the right places to yield the right definition. At no point did we, for example, identify derivation trees up to alpha, or identify reductions up to alpha. All of that work was done during the definition of \( \text{TERM} \), and the lifting of constructors and operations. The inductive definition is of the same form as we’ve always done, but now over equivalence classes of \( \text{CONCERETERM} \) rather than \( \text{CONCERETERM} \) themselves. But both of these entities are sets as far as set theory is concerned, so there’s no difference. If we added a type system to this language, the development would proceed much like it did for single-step reduction: typing rules would apply to \( \text{TERMS} \), injections would need to be appropriately placed, and that’s it. Even typing contexts \( \Gamma \) could be the same as always.

Similar identification processes can be applied to other things besides bound variable names. For instance, [Felleisen and Friedman][1989] presents a rigorous treatment of store locations, providing a semantics that abstracts away from the specific choice of store locations, and discusses the implications of that construction.

The common claim of “identifying terms up to alpha-equivalence” can be understood based on intuitions about what that might mean, but the devil is in the details, especially making sure that all the
relevant operations are well-defined for equivalence classes, and recognizing where representatives versus
class members appear in the development.

[RG: more gobbledygook] So we can represent entire alpha-equivalence classes of CONCRETE
 TERMS using a single CONCRETE TERM. Technically when we do so we are almost always referring to the class
not the individual CONCRETE TERM. However we do so using the notation of the original CONCRETE
TERM even though we are really talking about some TERM. Technically it takes a bunch of machinery to justify
this, but such is often the case when building up clean abstractions.

References

Hendrik Pieter Barendregt. The lambda calculus - its syntax and semantics, volume 103 of Studies in logic and

Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. Theoretical Computer


Christian Urban and Michael Norrish. A formal treatment of the Barendregt variable convention in rule in-
ductions. 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 Machin-

Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt’s variable convention in rule in-
ductions. In Proceedings of the 21st International Conference on Automated Deduction: Automated De-
10.1007/978-3-540-73595-3_4. URL https://doi.org/10.1007/978-3-540-73595-3_4.

A Proofs

Proposition 15 (Principle of Structural Induction for CONCRETE TERM). Let Φ be a property of terms t. Then
∀ t ∈ CONCRETE TERM. Φ(t) if:

1. (num) ∀ n ∈ N. Φ(n);
2. (plus) ∀ t₁, t₂ ∈ CONCRETE TERM. Φ(t₁) ∧ Φ(t₂) ⇒ Φ(t₁ + t₂);
3. (var) ∀ x ∈ VAR. Φ(x);
4. (lam) ∀ x ∈ VAR, t ∈ CONCRETE TERM. Φ(t) ⇒ Φ(λ x . t);
5. (app) ∀ t₁, t₂ ∈ CONCRETE TERM. Φ(t₁) ∧ Φ(t₂) ⇒ Φ(t₁ t₂);
6. (let) ∀ x ∈ VAR, t₁, t₂ ∈ CONCRETE TERM. Φ(t₁) ∧ Φ(t₂) ⇒ Φ(let x = t₁ in t₂).

Proposition 16 (Principle of Size Induction for CONCRETE TERM). Let Φ be a property of CONCRETE TERMS t. Then
the following holds:

∀ t₁ ∈ CONCRETE TERM. (∀ t₂ ∈ CONCRETE TERM. size(t₂) < size(t₁) ⇒ Φ(t₂)) ⇒ Φ(t₁) ⇒ ∀ t ∈ CONCRETE TERM. Φ(t)
Proposition 17 (Principle of Rule Induction for ($\sim_a$)). Let $\Phi$ be a property of $t_1 \sim_a t_2$ instances. Then \( \forall (t_1 \sim_a t_2), \Phi(t_1, t_2) \) if:

1. (anum) \( \forall n \in \mathbb{N}. \Phi(n, n); \)
2. (aplus) \( \forall (t_{11} \sim_a t_{12}), (t_{21} \sim_a t_{22}). \Phi(t_{11}, t_{21}) \land \Phi(t_{12}, t_{22}) \implies \Phi(t_{11} + t_{12}, t_{21} + t_{22}); \)
3. (aivar) \( \forall x \in \text{VAR}. \Phi(x, x); \)
4. (alam1) \( \forall x \in \text{VAR}, (t_1 \sim_a t_2). \Phi(t_1, t_2) \implies \Phi((\lambda x.t_1), (\lambda x.t_2)); \)
5. (alam2) \( \forall x_1, x_2, x_3 \in \text{VAR}, t_1, t_2 \in \text{CONCRETETERM}. \ x_1 \neq x_2 \land x_3 \notin (\text{Vars}(t_1) \setminus \{ x \}) \cup (\text{Vars}(t_2) \setminus \{ x \}) \land \Phi((\lambda x_3.x_1 \mapsto x_3[t_1]), (\lambda x_3.x_2 \mapsto x_3[t_2])) \implies \Phi((\lambda x_1.t_1), (\lambda x_2.t_2)); \)
6. (aapp) \( \forall (t_{11} \sim_a t_{12}), (t_{21} \sim_a t_{22}). \Phi(t_{11}, t_{21}) \land \Phi(t_{12}, t_{22}) \implies \Phi(t_{11} t_{12}, t_{21} t_{22}); \)
7. (alet1) \( \forall x \in \text{VAR}, (t_1 \sim_a t_2), (t_1 \sim_a t_2). \Phi(t_{11}, t_{21}) \land \Phi(t_{12}, t_{22}) \implies \Phi((\text{let } x = t_{11} \text{ in } t_{12}), (\text{let } x = t_{11} \text{ in } t_{12})); \)
8. (alet2) \( \forall x_1, x_2, x_3 \in \text{VAR}, t_{11}, t_{12}, t_{21}, t_{22} \in \text{CONCRETETERM}. \ x_1 \neq x_2 \land x_3 \notin (\text{Vars}(t_{12}) \setminus \{ x \}) \cup (\text{Vars}(t_{22}) \setminus \{ x \}) \land \Phi((\text{let } x_1 = t_{11} \text{ in } x_3[t_{12}]), (\text{let } x_1 = t_{21} \text{ in } x_3[t_{22}])) \implies \Phi((\text{let } x_1 = t_{11} \text{ in } t_{12}), (\text{let } x_2 = t_{21} \text{ in } t_{22})); \)

Proposition 18 (Principles of Inversion for ($\sim_a$), distinguishing $t_1$ in $t_1 \sim_a t_2$).

1. \( \forall n \in \mathbb{N}, t_2 \in \text{CONCRETETERM}. n \sim_a t_2 \implies t_2 = n; \)
2. \( \forall t_{11}, t_{12}, t_2 \in \text{CONCRETETERM}. t_{11} + t_{12} \sim_a t_2 \implies \exists t_{21}, t_{22} \in \text{CONCRETETERM}. t_2 = t_{21} + t_{22}; \)
3. \( \forall x \in \text{VAR}, t_2 \in \text{CONCRETETERM}. x \sim_a t_2 \implies t_2 = x; \)
4. \( \forall x_1 \in \text{VAR}, t_{11}, t_{12} \in \text{CONCRETETERM}. (\lambda x_1.t_{11}) \sim_a t_{12} \implies \exists x_2 \in \text{VAR}, t_{22} \in \text{CONCRETETERM}. t_2 = (\lambda x_2.t_{22}) \land ((x_2 = x_1 \land t_{11} \sim_a t_{22}) \lor (x_2 \neq x_1 \land \exists x_3 \in \text{VAR}. x_3 \notin \text{FV}(\lambda x_1.t_{11}) \land \text{FV}(\lambda x_2.t_{22}) \land (\lambda x_3.x_1 \mapsto x_3[t_{11}]) \sim_a (\lambda x_3.x_2 \mapsto x_3[t_{22}])))); \)
5. \( \forall t_{11}, t_{12}, t_2 \in \text{CONCRETETERM}. t_{11} t_{12} \sim_a t_2 \implies \exists t_{21}, t_{22} \in \text{CONCRETETERM}. t_2 = t_{21} t_{22}; \)

Proposition 19 (Refined inversion for ($\sim_a$) ($\lambda x_1.t_1 \sim_a (\lambda x_2.t_2)$)).

\( \forall x_1, x_2 \in \text{VAR}, t_1, t_2 \in \text{CONCRETETERM}. (\lambda x_1.t_1) \sim_a (\lambda x_2.t_2) \implies \forall x_3 \in \text{VAR}. x_3 \notin (\text{Vars}(t_1) \setminus \{ x_1 \}) \cup (\text{Vars}(t_2) \setminus \{ x_2 \}) \implies (\lambda x_3.x_1 \mapsto x_3[t_{11}]) \sim_a (\lambda x_3.x_2 \mapsto x_3[t_{22}]). \)

Proof. [RG: Fill Me In] I bet this is a pain in the ass, but it tells a useful story about the difference (and similarity) between shallow backward reasoning principles and other bespoke reasoning principles.

Strategy: when does naive renaming preserve alpha? Exploit that. \( \square \)

Proposition 1 ($\sim_a$) is an Equivalence Relation.

1. \( \forall t \in \text{CONCRETETERM}. t \sim_a t. \) In words, \( \sim_a \) is reflexive;
2. \( \forall t_1, t_2 \in \text{CONCRETETERM}. t_1 \sim_a t_2 \implies t_2 \sim_a t_1. \) In words, \( \sim_a \) is symmetric;
3. \( \forall t_1, t_2, t_3 \in \text{CONCRETETERM}. t_1 \sim_a t_2 \land t_2 \sim_a t_3 \implies t_1 \sim_a t_3. \) In words, \( \sim_a \) is transitive.

Proof of 1. Proof by structural induction on $t \in \text{CONCRETETERM}$. $\Phi(t) \equiv t \sim_a t$. 

Page 19
\textbf{Lemma 3 (num).} \( \forall n \in \mathbb{N}. \Phi(n) . \) \\
\textit{Proof.} Suppose \( n \in \mathbb{N} . \) Then by \((\alpha \text{num}) \) \( n \sim_a n . \)

\textbf{Lemma 4 (plus).} \( \forall t_1, t_2 \in \text{CONCRETETerm}. \Phi(t_1) \land \Phi(t_2) \Rightarrow \Phi(t_1 + t_2) . \) \\
\textit{Proof.} Suppose \( t_1, t_2 \in \text{CONCRETETerm}, t_1 \sim_a t_1, \) and \( t_2 \sim_a t_2 . \) Then apply \((\alpha \text{plus}) \) to the latter two to yield \( t_1 + t_2 \sim_a t_1 + t_2 . \)

\textbf{Lemma 5 (var).} \( \forall x \in \text{VAR}. \Phi(x) . \) \\
\textit{Proof.} Suppose \( x \in \text{VAR} . \) Then by \((\alpha \text{var}) \) \( x \sim_a x . \)

\textbf{Lemma 6 (lam).} \( \forall x \in \text{VAR}, t \in \text{CONCRETETerm}. \Phi(t) \Rightarrow \Phi(\lambda x.t) . \) \\
\textit{Proof.} Suppose \( x \in \text{VAR}, t \in \text{CONCRETETerm} \) and \( t \sim_a t . \) Then apply \((\alpha \text{lam}) \) to the latter to yield \( (\lambda x.t) \sim_a (\lambda x.t) . \)

\textbf{Lemma 7 (app).} \( \forall t_1, t_2 \in \text{CONCRETETerm}. \Phi(t_1) \land \Phi(t_2) \Rightarrow \Phi(t_1 \ t_2) . \) \\
\textit{Proof.} Suppose \( t_1, t_2 \in \text{CONCRETETerm}, t_1 \sim_a t_1, \) and \( t_2 \sim_a t_2 . \) Then apply \((\alpha \text{app}) \) to the latter two to yield \( t_1 \ t_2 \sim_a t_1 \ t_2 . \)

\textbf{Lemma 8 (let).} \( \forall x \in \text{VAR}, t_1, t_2 \in \text{CONCRETETerm}. \Phi(t_1) \land \Phi(t_2) \Rightarrow \Phi(\text{let } x = t_1 \ \text{in } t_2) . \) \\
\textit{Proof.} Suppose \( x \in \text{VAR}, t_1, t_2 \in \text{CONCRETETerm}, t_1 \sim_a t_1, \) and \( t_2 \sim_a t_2 . \) Then apply \((\alpha \text{let}) \) to the latter two to yield \( \text{let } x = t_1 \ \text{in } t_2 \sim_a \text{let } x = t_1 \ \text{in } t_2 . \)

\textit{Proof of 2.} Proof by rule induction on \((\cdot \sim_a \cdot) . \)

\( \Phi(t_1 \sim_a t_2) \equiv t_1 \sim_a t_2 \Rightarrow t_2 \sim_a t_2 . \)

\textbf{Left as an exercise for the reader} \\
\textit{Proof of 3.} Proof by size induction on \( t_2. \) \( \Phi(t_2) \equiv \forall t_1, t_3 \in \text{CONCRETETerm}. t_1 \sim_a t_2 \land t_2 \sim_a t_3 \Rightarrow t_1 \sim_a t_3 . \) Suppose \( t_2 \in \text{CONCRETETerm} \) and

\( \left( \forall t \in \text{CONCRETETerm}. \text{size}(t) < \text{size}(t_2) \Rightarrow \Phi(t) \right) \Rightarrow \Phi(t_2) . \)

Then it suffices to show \( \Phi(t_2) . \) So suppose \( t_1, t_3 \in \text{CONCRETETerm}, t_1 \sim_a t_2, \) and \( t_2 \sim_a t_3 . \) Then it suffices to show that \( t_1 \sim_a t_3 . \) We proceed by cases on the structure of \( t_2 \in \text{CONCRETETerm} . \)

\textbf{Case (t_2 = n). \ [RG: Fill Me In]}

\textbf{Case (\exists t_1, t_2 \in \text{CONCRETETerm}. t_2 = t_21 + t_22). \ [RG: Fill Me In]}

\textbf{Case (t_2 = x). \ [RG: Fill Me In]}

\textbf{Case (\exists x \in \text{VAR}, t_20 \in \text{CONCRETETerm}. t_2 = \lambda x.t_20). \ [RG: Fill Me In]}

\textbf{Case (\exists t_21, t_22 \in \text{CONCRETETerm}. t_2 = t_21 \ t_22). \ [RG: Fill Me In]}

\textbf{Case (\exists x \in \text{VAR}, t_21, t_22 \in \text{CONCRETETerm}. t_2 = \text{let } x = t_21 \ \text{in } t_22). \ [RG: Fill Me In]}

\textbf{Proposition 2 ([/\cdot:] is well-defined).} \textit{The equations in Defn. \[\frac{}{}\] describe a unique function.}

\( F \in \text{CONCRETETerm} \times \text{VAR} \times \text{CONCRETETerm} \rightarrow \text{CONCRETETerm} . \)
Proof. [RG: FILL ME IN!]

Proposition 3 (Representation).
1. \( \forall t \in \text{CONCRETE Term}. \ t \in A[t] \). In words, each representative is an element of the class it represents;
2. \( \forall t_1, t_2 \in \text{CONCRETE Term}. \ t_1 \sim_a t_2 \iff A[t_1] = A[t_2] \).
   In words, \( A[\cdot] \) exactly conceives the alpha-equivalence classes;
3. \( \forall t^\alpha \in \text{TERM}. \ \exists t \in t^\alpha. \ t^\alpha = A[t] \). In words, any element of \( t^\alpha \) can represent it;
4. \( \forall t^\alpha \in \text{TERM}. \ \exists t \in \text{CONCRETE Term}. \ t^\alpha = A[t] \). In words, \( A[\cdot] \) is surjective: every TERM can be represented by a CONCRETE TERM.

Proof.
1. Follows from Prop. 1(1).
2. Let \( t_1, t_2 \in \text{CONCRETE Term} \). We proceed by cases.
   \((\Rightarrow) \) Suppose \( t_1 \sim_a t_2 \). Then \( t \in A[t_1] \) Def. 8 \( \iff t \sim_a t_1 \iff t \sim_a t_2 \iff t \in A[t_2] \).
   \((\Leftarrow) \) Suppose \( A[t_1] = A[t_2] \). Then by part (1) \( t_1 \in A[t_1], t_2 \in A[t_2] \), and by extensionality, \( \forall t \in \text{CONCRETE Term}. \ t \in A[t_2] \iff t \in A[t_1] \). Therefore \( t_1 \in A[t_2] \) Def. 9 \( t_1 \sim_a t_2 \).
3. Let \( t^\alpha \in \text{TERM} \), and suppose \( t \in t^\alpha \). Then for \( t_0 \in \text{CONCRETE Term} \),
   \[ t_0 \in t^\alpha \defby 8 t_0 \sim_a t \defby 8 t_0 \in A[t] \).
4. Suppose \( t^\alpha \in \text{TERM} \). Then \( \exists t \in \text{CONCRETE Term}. t \in t^\alpha \). Then \( t_0 \in t^\alpha \defby 8 t_0 \sim_a t \defby 8 t_0 \in A[t] \).

Proposition 4 (Liftings).
1. \( \forall t_1, t_2 \in \text{CONCRETE Term}. \ A[t_1] \oplus_t A[t_2] = A[t_1 t_2] \);
2. \( \forall t_1, t_2 \in \text{CONCRETE Term}. \ A[t_1] \oplus_t A[t_2] = A[t_1 + t_2] \);
3. \( \forall x \in \text{VAR}. t \in \text{TERM}. \ \lambda^\alpha[x].A[t] = A[\lambda x.t] \).

Proof. [RG: FILL ME IN!]

Lemma 9 (Alpha-equivalence preserves size). If \( t_1 \sim_a t_2 \) then \( \text{size}(t_1) = \text{size}(t_2) \).

Proof. Induction on \( t_1 \sim_a t_2 \). [RG: FILL ME IN!]

Lemma 2 (Representational Inversion). Let \( t^\alpha \in \text{TERM} \). Then exactly one of the following is true:
1. \( \exists n \in \mathbb{Z}. \ t^\alpha = A[n] \);
2. \( \exists t_1, t_2 \in \text{CONCRETE Term}. \ t^\alpha = A[t_1 + t_2] \);
3. \( \exists x \in \text{VAR}. t^\alpha = A[x] \);
4. \( \exists x \in \text{VAR}. t \in \text{TERM}. \ t^\alpha = A[\lambda x.t] \);
5. \( \exists t_1, t_2 \in \text{CONCRETE Term}. \ t^\alpha = A[t_1 t_2] \);
6. \( \exists x \in \text{VAR}. t_1, t_2 \in \text{CONCRETE Term}. \ t^\alpha = A[\text{let } x = t_1 \text{ in } t_2] \).

Proof. [RG: FILL ME IN!]

Page 21
Proposition 5 (Structural Inversion). Let \( t^\alpha \in \text{TERM} \). Then exactly one of the following is true:

1. \( \exists n \in \mathbb{Z}. t^\alpha = n^\alpha; \)
2. \( \exists t_1^\alpha, t_2^\alpha \in \text{TERM}. t^\alpha = t_1^\alpha +^\alpha t_2^\alpha; \)
3. \( \exists x \in \text{VAR}. t^\alpha = x^\alpha; \)
4. \( \exists x \in \text{VAR}, t^\alpha \in \text{TERM}. t^\alpha = \lambda^\alpha[x]t^\alpha; \)
5. \( \exists t_1^\alpha, t_2^\alpha \in \text{TERM}. t^\alpha = t_1^\alpha \circ^\alpha t_2^\alpha; \)
6. \( \exists x \in \text{VAR}, t_1^\alpha, t_2^\alpha \in \text{TERM}. t^\alpha = \text{let}^\alpha(x, t_1^\alpha, t_2^\alpha). \)

Proof. [RG: Fill Me In]

Definition 20.

\[ \text{size}^\alpha : \text{TERM} \rightarrow \mathbb{N} \]
\[ \text{size}^\alpha(A[t]) = \text{size}(t) \]

The definiteness of above definition depends on Prop. 3(4) and Lm. 9.

Lemma 10 (Structural Size Equations).

1. \( \text{size}^\alpha(n^\alpha) = 1; \)
2. \( \text{size}^\alpha(t_1^\alpha +^\alpha t_2^\alpha) = 1 + \text{size}^\alpha(t_1^\alpha) + \text{size}^\alpha(t_2^\alpha); \)
3. \( \text{size}^\alpha(x^\alpha) = 1; \)
4. \( \text{size}^\alpha(\lambda^\alpha[x]t^\alpha) = 1 + \text{size}^\alpha(t^\alpha); \)
5. \( \text{size}^\alpha(t_1^\alpha \circ^\alpha t_2^\alpha) = 1 + \text{size}^\alpha(t_1^\alpha) + \text{size}^\alpha(t_2^\alpha); \)
6. \( \text{size}^\alpha(\text{let}^\alpha(x, t_1^\alpha, t_2^\alpha)) = 1 + \text{size}^\alpha(t_1^\alpha) + \text{size}^\alpha(t_2^\alpha); \)

Proof. [RG: Fill Me In]

Proposition 7 (Redefinition). \( \text{TERM} \subseteq \text{ST} \).

Proof. Suppose \( t^\alpha \in \text{TERM} \). We proceed by induction on \( \text{size}^\alpha(t^\alpha) \). By Lm. 2 exactly one of the following holds: [RG: Fix to use structural inversion, Prop. ??]

1. \( \exists n \in \mathbb{Z}. t^\alpha = A[n]. \) Then by (STn), \( t^\alpha \in \text{ST} \).
2. \( \exists t_1, t_2 \in \text{CONCRETE}\text{TERM}. t^\alpha = A[t_1 + t_2]. \) Then by Prop. 4, \( t^\alpha = A[t_1] +^\alpha A[t_2] \), and by Def. 9 \( A[t_1], A[t_2] \in \text{TERM} \). By the induction hypothesis, \( A[t_1], A[t_2] \in \text{ST} \) and by (ST+), \( t^\alpha \in \text{ST} \).
3. \( \exists x \in \text{VAR}. t^\alpha = A[x]. \) Then by (STx), \( t^\alpha \in \text{ST} \).
4. \( \exists x \in \text{VAR}, t \in \text{CONCRETE}\text{TERM}. t^\alpha = A[\lambda x.t]; \)
5. \( \exists t_1, t_2 \in \text{CONCRETE}\text{TERM}. t^\alpha = A[t_1 \circ t_2]; \)
6. \( \exists x \in \text{VAR}, t_1, t_2 \in \text{CONCRETE}\text{TERM}. t^\alpha = A[\text{let} x = t_1 \text{ in } t_2]. \)

Corollary 1. \( \text{TERM} = \text{ST} \).

Proof. Immediate consequence of Def. 11 (of ST) and Prop. 7.
Proposition 8 (Principle of Rule Induction on the structure of TERM). Let \( \Phi \) be a predicate on TERMS \( t^\alpha \). Then \( \Phi(t^\alpha) \) holds for all \( t^\alpha \in \text{TERM} \) if:

1. \( \forall n \in \mathbb{N}. \Phi(n^\alpha) \);
2. \( \forall t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha +^\alpha t_2^\alpha) \);
3. \( \forall x \in \text{VAR}. \Phi(x^\alpha) \);
4. \( \forall x \in \text{VAR}, t^\alpha \in \text{TERM}. \Phi(t^\alpha) \implies \Phi(\lambda^\alpha[x]t^\alpha) \);
5. \( \forall t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(t_1^\alpha \!@^\alpha t_2^\alpha) \);
6. \( \forall x \in \text{VAR}, t_1^\alpha, t_2^\alpha \in \text{TERM}. \Phi(t_1^\alpha) \land \Phi(t_2^\alpha) \implies \Phi(\text{let}^\alpha(x, t_1^\alpha, t_2^\alpha)) \).

Proof. [RG: Fill Me In]

Proposition 10 (Lifting). Let \( S \) be some set, and \( F : \text{CONCRETETERM} \to S \). Then if \( \forall t_1, t_2 \in \text{CONCRETETERM}. t_1 \sim_a t_2 \implies F(t_1) = F(t_2) \), then there is a unique function \( \text{F}^\alpha : \text{TERM} \to S \) such that \( \forall t \in \text{CONCRETETERM}. \text{F}^\alpha(A[t][]) = F(t) \).

Proof. [RG: Fill Me In]

Proposition 11. \( \forall t_1, t_2 \in \text{CONCRETETERM}. t_1 \sim_a t_2 \implies \text{FV}(t_1) = \text{FV}(t_2) \).

Proof. [RG: Fill Me In]

Proposition 12. \( \forall x \in \text{VAR}, t^\alpha \in \text{TERM}. \text{FV}^\alpha(\lambda^\alpha[x]t^\alpha) = \text{FV}^\alpha(t^\alpha) \setminus \{x\} \).

Proof. [RG: Fill Me In]

Proposition 13 (Capture-Avoiding Substitution Preserves Alpha-equivalence).

\[ \forall x \in \text{VAR}, t_{11}, t_{12}, t_{21}, t_{22} \in \text{CONCRETETERM}. t_{11} \sim_a t_{21} \land t_{12} \sim_a t_{22} \implies [t_{11}/x]t_{12} \sim_a [t_{21}/x]t_{22}. \]

Proof. [RG: Fill Me In]

Corollary 2 (Capture-Avoiding Substitution Lifts).

Let \( F : \text{CONCRETETERM} \times \text{VAR} \times \text{CONCRETETERM} \to \text{TERM} \) be defined by \( F = A[ ] \circ [ \cdot / \cdot ] \). Then \( \forall x \in \text{VAR}, t_{11}, t_{12}, t_{21}, t_{22} \in \text{CONCRETETERM}. t_{11} \sim_a t_{21} \land t_{12} \sim_a t_{22} \implies F(t_{11}, x, t_{12}) = F(t_{21}, x, t_{22}) \).

Proof. [RG: Fill Me In]