Chapter 22

Bound Variable Names Shouldn't Matter
a.k.a. Alpha-Equivalence Classes as Syntax
a.k.a. Barendregt's Variable Convention, Formally

> James Bond: "There seems to be some mistake. My name is—"
> Mr. Big: "Names is for tombstones, baby!"
> — Live and Let Die (1973).[1]

22.1 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.[2]

In [**?**], 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 $[t/x]t_0$ 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.[3]

[1] `https://www.youtube.com/watch?v=7pMWa33uVVE`
[2] Both capture-avoiding substitution and naïve substitution are defined below.
[3] The substitution notation has been modified from the original to match this exposition.

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.[4] So now we write $\lambda x.x \equiv \lambda y.y$, etcetera.

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. Moral 2.1.14: Using conventions 2.1.12 and 2.1.13, one can work with λ-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*.[5] 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 [**??**]. Teaching a computer to automatically identify bound variable names exposed some gaps, which also arise when working rigorously on paper.[6]

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 *representatives* 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.

[4] We define this term below under it's modern name α-equivalence.

[5] I know, I know: there are *two* conventions, but nonetheless they are typically evoked simultaneously. Barendregt was just trying to write a book, and it turns out that book has been highly influential.

[6] Urban et al.'s work uncovers additional subtleties with scaling up the automation of this approach. It is recommended reading.

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.

22.2 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 14 (Concrete Syntax)**.**

$$\begin{array}{l} x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad t \in \textsc{ConcreteTerm} \\ t \quad ::= \quad n \mid t + t \mid x \mid \lambda x.t \mid t\ t \mid \mathit{let}\ x = t\ \mathit{in}\ t \end{array}$$

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. 21), 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 abstract syntax, called Term, but we also use some of them to illustrate differences between ConcreteTerm and this new notion, by developing analogous notions in some cases, or explaining why no analogous notion exists in other cases.

Definition 15 (Free Variables)**.**

$$\begin{array}{rl} FV : \textsc{ConcreteTerm} \to \mathcal{P}(\textsc{Var}) & \\ FV(n) & = \emptyset \\ FV(t_1 + + t_2) & = FV(t_1) \cup FV(t_2) \\ FV(x) & = \{\, x \,\} \\ FV(\lambda x.t) & = FV(t) \setminus \{\, x \,\} \\ FV(t_1\ t_2) & = FV(t_1) \cup FV(t_2) \\ FV(\mathit{let}\ x == t_1\ \mathit{in}\ t_2) & = FV(t_1) \cup (FV(t_2) \setminus \{\, x \,\}) \end{array}$$

Definition 16 (Occurring Variables (i.e. Free or Bound Variables))**.**

$$\begin{array}{rl} Vars : \textsc{ConcreteTerm} \to \mathcal{P}(\textsc{Var}) & \\ Vars(n) & = \emptyset \\ Vars(t_1 + + t_2) & = Vars(t_1) \cup Vars(t_2) \\ Vars(x) & = \{\, x \,\} \\ Vars(\lambda x.t) & = \{\, x \,\} \cup Vars(t) \\ Vars(t_1\ t_2) & = Vars(t_1) \cup Vars(t_2) \\ Vars(\mathit{let}\ x == t_1\ \mathit{in}\ t_2) & = Vars(t_1) \cup Vars(t_2) \end{array}$$

Definition 17 (Naïve Substitution)**.**

$$
\begin{aligned}
[x \mapsto t] : \textit{ConcreteTerm} &\to \textit{ConcreteTerm} \\
[x \mapsto t]n &= n \\
[x \mapsto t](t_1 + + t_2) &= ([x \mapsto t]t_1) + + ([x \mapsto t]t_2) \\
[x \mapsto t]x &= t \\
[x \mapsto t]x_0 &= x_0 \textit{ if } x \neq x_0 \\
[x \mapsto t]\lambda x.t &= \lambda x.t \\
[x \mapsto t]\lambda x_0.t_0 &= \lambda x_0.[x \mapsto t]t_0 \textit{ if } x \neq x_0 \\
[x \mapsto t](t_1\ t_2) &= ([x \mapsto t]t_1)\ ([x \mapsto t]t_2) \\
[x \mapsto t](\textit{let } x == t_1 \textit{ in } t_2) &= \textit{let } x == [x \mapsto t]t_1 \textit{ in } t_2 \\
[x \mapsto t](\textit{let } x_0 == t_1 \textit{ in } t_2) &= \textit{let } x_0 = [x \mapsto t]t_1 \textit{ in } [x \mapsto t]t_2 \textit{ if } x \neq x_0
\end{aligned}
$$

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, $[y \mapsto x]\lambda z.y == \lambda z.x$. So naïve substitution does not preserve free variables. Capture-avoiding substitution (Defn. 20) 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 ConcreteTerms which says that they are identical but for their choice of bound variables. For many purposes, we consider such ConcreteTerms 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 ConcreteTerm with set-theoretic identity for our soon-to-be-defined set Term.

Definition 18 (Alphabetic equivalence (a.k.a. Alpha Equivalence))**.**

$\boxed{\sim_a \subseteq \textit{ConcreteTerm} \times \textit{ConcreteTerm}}$

$$\frac{}{n \sim_a n}\ (\alpha num) \qquad \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{t_{11} + + t_{12} \sim_a t_{21} + + t_{22}}\ (\alpha plus) \qquad \frac{}{x \sim_a x}\ (\alpha var) \qquad \frac{t_1 \sim_a t_2}{(\lambda x.t_1) \sim_a (\lambda x.t_2)}\ (\alpha lam1)$$

$$\frac{(\lambda x_3.[x_1 \mapsto x_3]t_1) \sim_a (\lambda x_3.[x_2 \mapsto x_3]t_2)}{(\lambda x_1.t_1) \sim_a (\lambda x_2.t_2)}\ (\alpha lam2) \quad \begin{array}{l} x_1 \neq x_2, \\ x_3 \notin (\mathit{Vars}(t_1) \setminus \{x_1\}) \cup (\mathit{Vars}(t_2) \setminus \{x_2\}) \end{array}$$

$$\frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{(t_{11}\ t_{12}) \sim_a (t_{21}\ t_{22})}\ (\alpha app) \qquad \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{\textit{let } x == t_{11} \textit{ in } t_{12} \sim_a \textit{let } x == t_{21} \textit{ in } t_{22}}\ (\alpha let1)$$

$$\frac{\textit{let } x_3 == t_{11} \textit{ in } [x_1 \mapsto x_3]t_{12} \sim_a \textit{let } x_3 == t_{21} \textit{ in } [x_2 \mapsto x_3]t_{22}}{\textit{let } x_1 == t_{11} \textit{ in } t_{12} \sim_a \textit{let } x_2 == t_{21} \textit{ in } t_{22}}\ (\alpha let2) \quad \begin{array}{l} x_1 \neq x_2, \\ x_3 \notin (\mathit{Vars}(t_{12}) \setminus \{x_1\}) \cup (\mathit{Vars}(t_{22}) \setminus \{x_2\}) \end{array}$$

The key rules underlying α-equivalence are (αlam2) (αlet2): 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 naïve substitution to specify renaming, and use the *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 $(\cdot \sim_a \cdot)$ is an *equivalence relation*.

Proposition 90 ($(\cdot \sim_a \cdot)$ is an Equivalence Relation)**.**

1. $\forall t \in \textit{ConcreteTerm}.\, t \sim_a t$. *In words,* $\sim_a$ *is reflexive;*

2. $\forall t_1, t_2 \in \textit{ConcreteTerm}.\, t_1 \sim_a t_2 \Rightarrow t_2 \sim_a t_1$. *In words,* $\sim_a$ *is symmetric;*

3. $\forall t_1, t_2, t_3 \in \textit{ConcreteTerm}.\, t_1 \sim_a t_2 \wedge t_2 \sim_a t_3 \Rightarrow t_1 \sim_a t_3$. *In words,* $\sim_a$ *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 ConcreteTerm, 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. 20 definitely describe a unique function starting at least a couple of different ways:[7]

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. 20;

2. We can define capture-avoiding substitution in terms of naïve substitution, exploiting the principle of recursion on the size of ConcreteTerms. We can then apply equational reasoning to prove that this function satisfies the equations in Defn. 20.

Either way, we have proven the existence of a function that satisfies the equations in Defn. 20, at which point we prove that at most one function satisfies these equations, proceeding by induction on the size of ConcreteTerms.

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

Definition 19 (Size of a ConcreteTerm)**.**

$$
\begin{aligned}
\textit{size} : \textit{ConcreteTerm} &\to \mathbb{N} \\
\textit{size}(n) &= 1 \\
\textit{size}(t_1 + + t_2) &= 1 + \textit{size}(t_1) + \textit{size}(t_2) \\
\textit{size}(x) &= 1 \\
\textit{size}(\lambda x.t) &= 1 + \textit{size}(t) \\
\textit{size}(t_1\ t_2) &= 1 + \textit{size}(t_1) + \textit{size}(t_2) \\
\textit{size}(\textit{let}\ x == t_1\ \textit{in}\ t_2) &= 1 + \textit{size}(t_1) + \textit{size}(t_2)
\end{aligned}
$$

[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 20 (Capture-Avoiding Substitution)**.**

$$
\begin{aligned}
[\cdot/\cdot]\cdot : \mathit{ConcreteTerm} \times \mathit{Var} \times \mathit{ConcreteTerm} &\to \mathit{ConcreteTerm} \\
[t/x]b &= b \\
[t/x]x &= t \\
[t/x]x_0 &= x_0 \ \mathit{if}\ x \neq x_0 \\
[t/x](\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3) &= \mathit{if}\ [t/x]t_1\ \mathit{then}\ [t/x]t_2\ \mathit{else}\ [t/x]t_3 \\
[t/x](\mathit{let}\ x == t_1\ \mathit{in}\ t_2) &= \mathit{let}\ x == [t/x]t_1\ \mathit{in}\ t_2 \\
[t/x](\mathit{let}\ x_0 == t_1\ \mathit{in}\ t_2) &= \mathit{let}\ x_1 == [t/x]t_1\ \mathit{in}\ [t/x][x_1/x_0]t_2 \\
&\quad \mathit{if}\ x \neq x_0 \\
&\quad \mathit{where}\ x_1 = \begin{cases} x_0 & x \notin \mathit{FV}(t_2) \vee x_0 \notin \mathit{FV}(t) \\ \min_{x \in \mathit{Var}}(x \notin \mathit{Vars}(t) \cup \mathit{Vars}(t_2))) & \mathit{otherwise} \end{cases}
\end{aligned}
$$

Notice that we are using Haskell Curry's "trick" (described in the appendix of **?**) 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 91 ($[\cdot/\cdot]\cdot$ is well-defined)**.** *The equations in Defn. 20 describe a unique function.*

$$F \in \mathit{ConcreteTerm} \times \mathit{Var} \times \mathit{ConcreteTerm} \to \mathit{ConcreteTerm}.$$

22.3 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 21 (Terms as Alpha-equivalence Classes)**.**

$$
\begin{aligned}
t^\alpha \in \mathit{Term} &= \mathit{ConcreteTerm}/\sim_a \\
&= \left\{ S \in \mathcal{P}(\mathit{ConcreteTerm}) \;\middle|\; \begin{pmatrix} \exists t \in \mathit{ConcreteTerm}.\, t \in S \end{pmatrix} \wedge \\ \begin{pmatrix} \forall t \in \mathit{ConcreteTerm}.\, t \in S \Leftrightarrow \\ \quad \forall t_0 \in \mathit{ConcreteTerm}.\, t_0 \in S \Leftrightarrow t_0 \sim_a t \end{pmatrix} \right\}
\end{aligned}
$$

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 [**?**].

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

set of ConcreteTerms? 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 ConcreteTerm. 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 ConcreteTerm, 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^H^H^H^H notation for talking about them. The takeaway is: ultimately we will care about the API, not the junk underneath that we use to demonstrate that the API is "implementable".

Let's start on building up that interface. For our purposes, it will be very important to map ConcreteTerms 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 22 (Injection)**.**

$$\mathcal{A}[\![\cdot]\!] : \mathit{ConcreteTerm} \to \mathit{Term}$$
$$\mathcal{A}[\![t]\!] = \{\, t_0 \in \mathit{ConcreteTerm} \mid t \sim_a t_0 \,\}$$

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 ConcreteTerms that *represent* them, leaving any necessary injection operations implicit.[8]

Proposition 92 (Representation)**.**

1. $\forall t \in \mathit{ConcreteTerm}.\, t \in \mathcal{A}[\![t]\!]$. *In words, each representative is an element of the class it represents;*
2. $\forall t_1, t_2 \in \mathit{ConcreteTerm}.\, t_1 \sim_a t_2 \Leftrightarrow \mathcal{A}[\![t_1]\!] = \mathcal{A}[\![t_2]\!]$.
   In words, $\mathcal{A}[\![\cdot]\!]$ exactly conveys the alpha-equivalence classes;
3. $\forall t^\alpha \in \mathit{Term}.\, \forall t \in t^\alpha.\, t^\alpha = \mathcal{A}[\![t]\!]$. *In words, any element of t^α can represent it;*
4. $\forall t^\alpha \in \mathit{Term}.\, \exists t \in \mathit{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t]\!]$. *In words, $\mathcal{A}[\![\cdot]\!]$ is surjective: every Term can be* represented *by a ConcreteTerm.*

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 ConcreteTerms as representatives of the underlying alpha-equivalence classes. In short, we can implicitly treat a ConcreteTerm as whichever alpha-equivalent analogue is most convenient to operate with at that moment. Part 1) corresponds to Convention 2.1.12,

[8] 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 ConcreteTerms as representatives (but we'll steal their notation).

and Part 2) corresponds to Variable Convention 2.1.13. Part 2 in particular is meant to once-and-for-all introduce side-conditions on expressions that can now be assumed implicitly. The idea is that so long 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:

$$\frac{}{(\lambda x.t)\ v \longrightarrow [v/x]t}\ (\text{sapp})$$

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

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

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

$$\frac{}{t_1^\alpha \longrightarrow t_2^\alpha}\ (\text{sapp}) \quad \begin{array}{l} \exists x \in \textsc{Var}, t \in \textsc{Term}, v \in \textsc{Value}. \\ (BV(\lambda x.t) \cap FV(v) = \emptyset) \wedge (BV(v) \cap FV(\lambda x.t) = \emptyset) \wedge \\ \qquad (t_1^\alpha = \mathcal{A}[\![(\lambda x.t)\ v]\!]) \wedge (t_2^\alpha = \mathcal{A}[\![[v/x]t]\!]). \end{array}$$

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 ConcreteTerms and the relationships that hold between *them*. Two Terms are related by reduction only if constraints on two ConcreteTerms, which inject to them, can be satisfied. So since the description is primarily in terms of ConcreteTerms, 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? **?** demonstrate this in action.

One key principle behind this quotienting is that the most relevant reasoning principles for ConcreteTerm 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 ConcreteTerm

Lemma 9 (Inversion)**.** *Let $t \in$ ConcreteTerm. Then exactly one of the following is true:*

1. $\exists! n \in \mathbb{Z}.\, t = n$;
2. $\exists! t_1, t_2 \in \textit{ConcreteTerm}.\, t = t_1 + t_2$;
3. $\exists! x \in \textit{Var}.\, t = x$;
4. $\exists! x \in \textit{Var}, t_0 \in \textit{ConcreteTerm}.\, t = \lambda x.t_0$;
5. $\exists! t_1, t_2 \in \textit{ConcreteTerm}.\, t = t_1\ t_2$;
6. $\exists! x \in \textit{Var}, t_1, t_2 \in \textit{ConcreteTerm}.\, t = \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$.

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

Lemma 10 (Representational Inversion)**.** *Let $t^\alpha \in$ Term. Then exactly one of the following is true:*

1. $\exists! n \in \mathbb{Z}.\, t^\alpha = \mathcal{A}[\![n]\!]$;
2. $\exists! t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1 + t_2]\!]$;
3. $\exists! x \in \textsc{Var}.\, t^\alpha = \mathcal{A}[\![x]\!]$;
4. $\exists x \in \textsc{Var}, t \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\lambda x.t]\!]$;
5. $\exists! t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1\ t_2]\!]$;
6. $\exists x \in \textsc{Var}, t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\mathit{let}\ x = t_1\ \mathit{in}\ t_2]\!]$.

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

22.4 Beyond Barendregt: Alpha-equivalent Syntax

Term "constructors" Now we deal with the pesky $\mathcal{A}[\![\cdot]\!]$, or by hiding it behind some more abstracting machinery. In particular we lift ConcreteTerm constructors to 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 ConcreteTerm t to denote a Term t^α.

Definition 23 (Term Operators)**.**

$$\begin{aligned}
&\cdot^\alpha : (\mathbb{Z} \cup \textit{Var}) \to \textit{Term} \\
&n^\alpha = \mathcal{A}[\![n]\!] \\
&x^\alpha = \mathcal{A}[\![x]\!]
\end{aligned}$$

$$\begin{aligned}
&\bullet^\alpha : \textit{Term} \times \textit{Term} \to \textit{Term} \\
&t_1^\alpha \bullet^\alpha t_2^\alpha = \{\, (t_1\ t_2) \in \textit{Term} \mid t_1 \in t_1^\alpha, t_2 \in t_2^\alpha \,\}
\end{aligned}$$

$$\begin{aligned}
&+^\alpha : \textit{Term} \times \textit{Term} \to \textit{Term} \\
&t_1^\alpha +^\alpha t_2^\alpha = \{\, t_1 + + t_2 \in \textit{Term} \mid t_1 \in t_1^\alpha, t_2 \in t_2^\alpha \,\}
\end{aligned}$$

$$\begin{aligned}
&\lambda^\alpha : \textit{Var} \to \textit{Term} \to \textit{Term} \\
&\lambda^\alpha[x]t^\alpha = \{\, \lambda x.t \in \textit{Term} \mid t \in t^\alpha \,\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{let}^\alpha : \textit{Var} \times \textit{Term} \times \textit{Term} \to \textit{Term} \\
&\mathit{let}^\alpha(x, t_1^\alpha, t_2^\alpha) = \{\, \mathit{let}\ x = t_1\ \mathit{in}\ t_2 \in \textit{Term} \mid t_1 \in t_1^\alpha, t_2 \in t_2^\alpha \,\}
\end{aligned}$$

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 93 (Liftings)**.**

1. $\forall t_1, t_2 \in \textsc{ConcreteTerm}.\, \mathcal{A}[\![t_1]\!] \bullet^\alpha \mathcal{A}[\![t_2]\!] = \mathcal{A}[\![t_1\ t_2]\!]$;
2. $\forall t_1, t_2 \in \textsc{ConcreteTerm}.\, \mathcal{A}[\![t_1]\!] +^\alpha \mathcal{A}[\![t_2]\!] = \mathcal{A}[\![t_1 + t_2]\!]$;
3. $\forall x \in \textsc{Var}, t \in \textsc{Term}.\, \lambda^\alpha[x]\mathcal{A}[\![t]\!] = \mathcal{A}[\![\lambda x.t]\!]$.

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 ConcreteTerms) justify definitions of the given functions (which ultimately are about 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 $\cdot^\alpha$, which embeds numbers and variables into our TERM classes. This concept makes direct sense in CONCRETETERM, where a variable reference x is *not* the same thing as a binder λx. You can think of the variable reference as being $\mathsf{var}[x]$, an operator *indexed* on a particular variable, and a number being $\mathsf{num}[n]$, an operator indexed on a particular number. So $\mathbb{N}$ and VAR are injected into CONCRETETERM, even though our notation makes that implicit. Our approach to TERM makes it explicit.

Now armed with operators that lift our term construction principles from CONCRETETERM to TERM, we can analogously lift our TERM analysis principles:

Proposition 94 (Structural Inversion)**.** *Let $t^\alpha \in$ TERM. Then exactly one of the following is true:*

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

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 TERMs into *unique* constituents. A TERM $\mathcal{A}[\![\lambda x.t]\!]$ decomposes into *many* possible VARs x and TERMs t^α. 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 TERM, and is sometimes called its *finite support* (a term from nominal logic, a theory for modeling binders developed by Andrew Pitts and colleagues **?**). This leads to extremely useful, more powerful principles for terms with bound variables.

Proposition 95 (Structural Inversion with Finite Support)**.** *Let $t^\alpha \in$ TERM. Then*

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

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 $\mathcal{X}$.

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

22.4.1 Induction On Alpha-Equivalence Classes

The direct definition of 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 TERM, in painful detail. First, following the usual strategy, we define "some structural term subset" ST of TERM (hint, it's not just *any* subset).

Definition 24 (Structural Term)**.** $\boxed{ST \subseteq \text{TERM}}$

$$\frac{}{n^\alpha \in ST}\ (STn) \qquad \frac{t_1^\alpha \in ST \quad t_2^\alpha \in ST}{t_1^\alpha +^\alpha t_2^\alpha \in ST}\ (ST+) \qquad \frac{}{x^\alpha \in ST}\ (STx) \qquad \frac{t^\alpha \in ST}{\lambda^\alpha[x]t^\alpha \in ST}\ (ST\lambda)$$

$$\frac{t_1^\alpha \in ST \quad t_2^\alpha \in ST}{t_1^\alpha \bullet^\alpha t_2^\alpha \in ST}\ (ST@) \qquad \frac{t_1^\alpha \in ST \quad t_2^\alpha \in ST}{let^\alpha(x, t_1^\alpha, t_2^\alpha) \in ST}\ (STlet)$$

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 96 (Redefinition)**.** $\textsc{Term} \subseteq ST$.

Corollary 2. $\textsc{Term} = ST$.

And voila! By extensionality Term inherits ST's induction principle:

Proposition 97 (Principle of Rule Induction on the structure of Term)**.** *Let* Φ *be a predicate on* Terms t^α*. Then* $\Phi(t^\alpha)$ *holds for all* $t^\alpha \in \textsc{Term}$ *if:*

1. $\forall n \in \mathbb{N}.\Phi(n^\alpha)$;
2. $\forall t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha +^\alpha t_2^\alpha)$;
3. $\forall x \in \textsc{Var}.\Phi(x^\alpha)$;
4. $\forall x \in \textsc{Var}, t^\alpha \in \textsc{Term}.\, \Phi(t^\alpha) \Rightarrow \Phi(\lambda^\alpha[x]t^\alpha)$;
5. $\forall t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha \bullet^\alpha t_2^\alpha)$;
6. $\forall x \in \textsc{Var}, t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(\mathit{let}^\alpha(x, t_1^\alpha, t_2^\alpha))$.

The above inductive definition and corresponding induction principle are appealing: they make Term look and feel a lot like ConcreteTerm, providing analogous reasoning principles. However, we sometimes want stronger reasoning principles for Term, which give us even stronger "induction hypotheses" when performing proof by induction.

To explain, consider the induction lemma induced for Λ, Prop. 97.4. It says that having assumed a particular variable x and a particular t^α and the property Φ holding for that particular t^α, we must prove that

$\Phi(\lambda^\alpha[x](t^\alpha)$ holds. The unsatisfying thing about this lemma is that a term $\lambda^\alpha[x](t^\alpha)$ can be represented by *many different* $x', t^{\alpha\prime}$ pairs, such that $\lambda^\alpha[x'](t^{\alpha\prime}) = \lambda^\alpha[x](t^\alpha)$, 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 $\Phi(t^{\alpha\prime})$, only $\Phi(t^\alpha)$. 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 Term that is motivated by the structure that we have superimposed on the set.

$\boxed{\sqsubset \subseteq \textsc{Term} \times \textsc{Term}}$ **α-Structural Order**

$$t_1^\alpha \sqsubset t_1^\alpha \bullet^\alpha t_2^\alpha \qquad t_2^\alpha \sqsubset t_1^\alpha \bullet^\alpha t_2^\alpha \qquad t^\alpha \sqsubset \lambda^\alpha[x](\widehat{t}) \qquad t_1^\alpha \sqsubset \mathit{let}^\alpha(x, t_1^\alpha, t_2^\alpha) \qquad t_2^\alpha \sqsubset \mathit{let}^\alpha(x, t_1^\alpha, t_2^\alpha)$$

To prove well-foundnedness, it suffices to show that $t_1^\alpha \sqsubset t_2^\alpha$ implies $\mathit{size}(t_1^\alpha) < \mathit{size}(t_2^\alpha)$ given an appropriate size function for Term. We discuss how to use the size function for ConcreteTerm to define the analogous function for Term in the next section.

The $\sqsubset$ relation induces a principle of well-founded induction on 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^α such that $t_1^\alpha \sqsubset t_2^\alpha$ as the premises for the rule with conclusion t_2^α. This means that the instances of the rule for a lambda abstraction will have the form:

$$\frac{\{\, t^\alpha \in \textsc{Term} \mid \exists x \in \textsc{Var}.\, \lambda^\alpha[x](t^\alpha) = \lambda^\alpha[x_0]t_0^\alpha \,\}}{\lambda^\alpha[x_0]t_0^\alpha}\ (\sqsubset \lambda)$$

Thus, the premise includes *all* candidate Term bodies. We can then take the corresponding induction principle and explicitly distinguish the structure of Term to yield the following more expressive one.

Proposition 98 (Principle of Rule Induction on the $\alpha-$structure of Term)**.** *Let Φ be a predicate on Terms t^α. Then $\Phi(t^\alpha)$ holds for all $t^\alpha \in$ Term if:*

1. $\forall n \in \mathbb{N}.\Phi(n^\alpha)$;
2. $\forall t_1^\alpha, t_2^\alpha \in \mathit{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha +^\alpha t_2^\alpha)$;
3. $\forall x \in \mathit{Var}.\Phi(x^\alpha)$;
4. $\forall x \in \mathit{Var}, t^\alpha \in \mathit{Term}.\, (\forall x_0 \in \mathit{Var}, t_0^\alpha \in \mathit{Term}.\, \lambda^\alpha[x_0]t_0^\alpha = \lambda^\alpha[x]t^\alpha \Rightarrow \Phi(t_0^\alpha)) \Rightarrow \Phi(\lambda^\alpha[x]t^\alpha)$;
5. $\forall t_1^\alpha, t_2^\alpha \in \mathit{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha \bullet^\alpha t_2^\alpha)$;
6. $\forall x \in \mathit{Var}, t_1^\alpha, t_2^\alpha \in \mathit{Term}.\, \Phi(t_1^\alpha) \wedge (\forall x_0 \in \mathit{Var}, t_0^\alpha \in \mathit{Term}.\, let^\alpha(x_0, t_1^\alpha, t_0^\alpha) = let^\alpha(x, t_1^\alpha, t_2^\alpha) \Rightarrow \Phi(t_0^\alpha)) \Rightarrow \Phi(let^\alpha(x, t_1^\alpha, t_2^\alpha))$.

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 α-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?

22.4.2 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 99 (Lifting)**.** *Let S be some set, and $F : \mathit{ConcreteTerm} \to S$. Then if $\forall t_1, t_2 \in \mathit{ConcreteTerm}.\, t_1 \sim_a t_2 \Rightarrow F(t_1) = F(t_2)$, then there is a unique function $F^\alpha : \mathit{Term} \to S$ such that $\forall t \in \mathit{ConcreteTerm}.\ F^\alpha\ (\mathcal{A}[\![t]\!]) = F(t)$.*

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 100. $\forall t_1, t_2 \in \mathit{ConcreteTerm}.\, t_1 \sim_a t_2 \Rightarrow FV(t_1) = FV(t_2)$.

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 101. $\forall x \in \mathit{Var}, t^\alpha \in \mathit{Term}.\ FV^\alpha\ (\lambda^\alpha[x]t^\alpha) = FV^\alpha(t^\alpha) \setminus \{\, x \,\}$.

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 25 (Bound Variables)**.**

$$
\begin{aligned}
BV : \mathit{ConcreteTerm} &\to \mathcal{P}(\mathit{Var}) \\
BV(n) &= \emptyset \\
BV(t_1 + +t_2) &= BV(t_1) \cup BV(t_2) \\
BV(x) &= \emptyset \\
BV(\lambda x.t) &= BV(t) \cup \{\, x \,\} \\
BV(t_1\ t_2) &= BV(t_1) \cup BV(t_2) \\
BV(\mathsf{let}\ x == t_1\ \mathit{in}\ t_2) &= BV(t_1) \cup (BV(t_2) \cup \{\, x \,\})
\end{aligned}
$$

Note that λy.y $\sim_a$ λx.x but $BV(\lambda y.y) = \{\, y \,\} \neq \{\, x \,\} = BV(\lambda x.x)$. In fact, $\mathcal{A}[\![\lambda x.x]\!] = \{\, \lambda x.x \mid x \in \mathrm{Var} \,\}$, 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!

22.4.3 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, Naïve 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 naïve substitution breaks down, and where it succeeds].

However, *capture-avoiding* substitution does remain meaningful after we lift our syntax from ConcreteTerm to Term!

Proposition 102 (Capture-Avoiding Substitution Preserves Alpha-equivalence)**.**

$$\forall x \in \mathit{Var}, t_{11}, t_{12}, t_{21}, t_{22} \in \mathit{ConcreteTerm}.\, t_{11} \sim_a t_{21} \wedge t_{12} \sim_a t_{22} \Rightarrow [t_{11}/x]t_{12} \sim_a [t_{21}/x]t_{22}.$$

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 ConcreteTerms and one Var. Technically, capture-avoiding substitution preserves alpha equivalence in several ways:

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

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 ConcreteTerms 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 3 (Capture-Avoiding Substitution Lifts)**.**
Let $F : \mathit{ConcreteTerm} \times \mathit{Var} \times \mathit{ConcreteTerm} \to \mathit{Term}$ *be defined by* $F = \mathcal{A}[\![\cdot]\!] \circ [\cdot/\cdot]\cdot$. *Then* $\forall x \in \mathit{Var}, t_{11}, t_{12}, t_{21}, t_{22} \in \mathit{ConcreteTerm}.\, t_{11} \sim_a t_{21} \wedge t_{12} \sim_a t_{22} \Rightarrow F(t_{11}, x, t_{12}) = F(t_{21}, x, t_{22})$.

Now we can apply Prop 99 to $(\mathcal{A}[\![\cdot]\!] \circ [\cdot/\cdot]\cdot)$ to lift capture-avoiding substitution.

Definition 26 (Lifted Substitution)**.**

$$[\cdot/\cdot]^\alpha\cdot : \mathit{Term} \times \mathit{Var} \times \mathit{Term} \to \mathit{Term}$$
$$[\mathcal{A}[\![t_1]\!]/x]^\alpha \mathcal{A}[\![t_2]\!] = \mathcal{A}[\![[t_1/x]^\alpha t_2]\!].$$

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 naïve substitution.

Proposition 103 ((Capture-avoiding) Substitution, Recursive-ishly)**.** *The* $[\cdot/\cdot]^\alpha\cdot$ *function is the function in* $\mathit{Term} \times \mathit{Var} \times \mathit{Term}$ *uniquely described by the following equations:*

$$\begin{aligned}
[t^\alpha/x]^\alpha n &= n \\
[t^\alpha/x]^\alpha (t_1^\alpha +^\alpha t_2^\alpha) &= ([t^\alpha/x]^\alpha t_1^\alpha) +^\alpha ([t^\alpha/x]^\alpha t_2^\alpha) \\
[t^\alpha/x]^\alpha x^\alpha &= t^\alpha \\
[t^\alpha/x]^\alpha {x_0}^\alpha &= {x_0}^\alpha \quad \textit{if } x_0 \neq x \\
[t^\alpha/x]^\alpha (t_1^\alpha \bullet^\alpha t_2^\alpha) &= ([t^\alpha/x]^\alpha t_1^\alpha) \bullet^\alpha ([t^\alpha/x]^\alpha t_2^\alpha) \\
[t^\alpha/x]^\alpha let^\alpha(x_0, t_1^\alpha, t_2^\alpha) &= let^\alpha(x_0, ([t^\alpha/x]^\alpha t_1^\alpha), ([t^\alpha/x]^\alpha t_2^\alpha)) \textit{ if } x \neq x_0 \\
[t^\alpha/x]^\alpha (\lambda^\alpha[x_0]t_0^\alpha) &= \lambda^\alpha[x_0]([t^\alpha/x]^\alpha t_0^\alpha) \textit{ if } x \neq x_0
\end{aligned}$$

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.

22.4.4 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 **?**)

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 CONCRETETERMs, our old notion, by taking the concept of alpha-equivalence to turn mere equivalence relationships into set-theoretic identities. Then, we tamed these blobby sets by building up a more pleasant "structural" interface to them: we constructed operators like λ^α and $\bullet^\alpha$, which are directly analogous to our constructors for CONCRETETERMs. 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 CONCRETETERM*. So that's exactly what we're going to do: from now on we we change notation, using *the original notation for CONCRETETERMs to refer to our constructed operators for TERMs*! We don't care about CONCRETETERMs 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 27 (Abstract Syntax)**.**

$$\begin{array}{l} x \in \mathit{VAR}, \quad n \in \mathbb{Z}, \quad t \in \mathit{TERM} \\ t \quad ::= \quad n \mid t + t \mid x \mid \lambda x.t \mid t\ t \mid \mathit{let}\ x = t\ \mathit{in}\ t \end{array}$$

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 CONCRETETERM 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 CONCRETETERM. 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. 27 is a phenomenal compression of the entire construction process that we described above, corresponding in essence to Def. 23, which gives construction principles, Prop. 94, which gives destruction principles, and Prop. 97, 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 CONCRETETERM (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 Term, thanks to the fact that it is not a simple inductive definition, and its inductive characterization and corresponding principle of induction is not the same as your typical syntactic definition. In particular, Defn. 24 is not a *deterministic* inductive definition: because of choice of bound variables, a member of 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 28 (Substitution (with abstract Terms))**.**
$[\cdot/\cdot]\cdot : \mathit{Term} \times \mathit{Var} \times \mathit{Term} \to \mathit{Term}$

$$
\begin{aligned}
[t/x]n &= n \\
[t/x](t_1 + + t_2) &= ([t/x]t_1) + + ([t/x]t_2) \\
[t/x]x &= t \\
[t/x]x_0 &= x_0 \quad \mathit{if}\ x_0 \neq x \\
[t/x](t_1\ t_2) &= ([t/x]t_1)\ ([t/x]t_2) \\
[t/x]\mathit{let}\ x_0 == t_1\ \mathit{in}\ t_2) &= \mathit{let}\ x_0 == ([t/x]t_1)\ \mathit{in}\ ([t/x]t_2)\ \mathit{if}\ x \neq x_0 \\
[t/x]\lambda x_0.t_0 &= \lambda x_0.[t/x]t_0\ \mathit{if}\ x \neq x_0
\end{aligned}
$$

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 ConcreteTerm to Term.

In particular, if we were to compare this to the Principle of Definition by Recursion for ConcreteTerm, 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 Term by appealing back to ConcreteTerm 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 *α-structural recursion* for Term. Such a principle should allow us to state the definition of functions over 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]

22.5 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 29 (Dynamic Semantics Artifacts)**.**

$$
\begin{array}{l}
v \in \textit{Value} \subseteq \textit{Pgm}, \quad F \in \textit{Frame}, \quad r \in \textit{Redex} \subseteq \textit{Pgm} \\
f \in \textit{Faulty} \subseteq \textit{Redex}, \quad \textit{err} \in \textit{Error}, \quad c \in \textit{Config} \\
p \in \textit{Pgm} = \{\, t \in \textit{Term} \mid \textit{FV}(t) = \emptyset \,\}, \quad \textit{Obs} = \{\, \textit{procedure}, \infty \,\} \cup \mathbb{Z} \cup \textit{Error} \\
\begin{array}{rcll}
v & ::= & n \mid \lambda x.t & \\
F & ::= & [\,]\ p \mid v\ \square \mid \textit{let}\ x = \square\ \textit{in}\ t \mid \square + p \mid v + \square & \\
r & ::= & v\ v \mid \textit{let}\ x = v\ \textit{in}\ t \mid v + v & \\
f & ::= & v_1\ v_2 & v_1 \notin \{\, \lambda x.t \in \textit{Term} \,\} \\
 & \mid & v_1 + + v_2 & \{\, v_1, v_2 \,\} \not\subseteq \mathbb{Z} \\
\textit{err} & ::= & \textit{mismatch} & \\
c & ::= & p \mid \textit{err} &
\end{array}
\end{array}
$$

We'll begin our unpacking with the defintiion of Pgm, the set of programs. Don't let the metavariable t fool you, we really mean t^α. And we're not using the free variable function on ConcreteTerm either. We've just dropped the $\cdot^\alpha$. So really:

$$\textsc{Pgm} = \{\, t^\alpha \in \textsc{Term} \mid FV^\alpha(t^\alpha) = \emptyset \,\}$$

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:

$\boxed{\textsc{Value} \subseteq \textsc{Pgm}}$

$$\frac{}{n^\alpha \in \textsc{Value}} \qquad \frac{}{\lambda^\alpha[x]t^\alpha \in \textsc{Value}}\ t^\alpha \in \textsc{Term}$$

The second rule is the interesting one. Since Value is defined as a subset of Pgm, we automatically get the constraint that $\lambda^\alpha[x]t^\alpha \in \textsc{Pgm}$ 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 Term or ConcreteTerm 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 30 (Single-step Reduction)**.** $\boxed{\longrightarrow \subseteq \textit{Pgm} \times \textit{Config}}$

$$\frac{}{(\lambda x.t)\ v \longrightarrow [v/x]t}\ (\textit{sapp}) \qquad \frac{}{\textit{let}\ x == v\ \textit{in}\ t \longrightarrow [v/x]t}\ (\textit{slet}) \qquad \frac{n_3 = n_1 + n_2}{n_1 + + n_2 \longrightarrow n_3}\ (\textit{splus})$$

$$\frac{}{f \longrightarrow \textit{mismatch}}\ (\textit{serr}) \qquad \frac{p_1 \longrightarrow p_2}{F[p_1] \longrightarrow F[p_2]}\ (\textit{Fstep}) \qquad \frac{p_1 \longrightarrow \textit{err}}{F[p_1] \longrightarrow \textit{err}}\ (\textit{Ferr})$$

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

$$\frac{}{(\lambda^\alpha[x]t^\alpha) \bullet^\alpha v \longrightarrow [v/x]^\alpha t^\alpha}\ (\text{sapp})$$

Notice the use of 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:

$$\frac{}{\mathcal{A}[\![(\lambda x.t_1)\ t_2]\!] \longrightarrow \mathcal{A}[\![[t_2/x]t_2]\!]}\ (\text{sapp}) \quad t_2 \in v$$

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 ConcreteTerms. This is arguably a violation of abstraction boundaries, but then using $\mathcal{A}[\![\cdot]\!]$ 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 $\mathcal{A}[\![\cdot]\!]$). 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 31 (Divergence)**.** $\boxed{\stackrel{\infty}{\longrightarrow} \subseteq \mathit{PGM}}$

$$\frac{p_1 \longrightarrow p_2 \quad p_2 \stackrel{\infty}{\longrightarrow}}{p_1 \stackrel{\infty}{\longrightarrow}}$$

Evaluation benefits from some treatment.

Definition 32 (Evaluation)**.**

$$\begin{array}{rr} \multicolumn{2}{c}{\mathit{eval} : \mathit{PGM} \to \mathit{OBS}} \\ \mathit{eval}(p) = n & \text{if } p \longrightarrow^* n \\ \mathit{eval}(p) = \textsf{procedure} & \text{if } p \longrightarrow^* \lambda x.t \\ \mathit{eval}(p) = \mathit{err} & \text{if } p \longrightarrow^* \mathit{err} \\ \mathit{eval}(p) = \infty & \text{if } p \stackrel{\infty}{\longrightarrow} \end{array}$$

Since 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.

$$\begin{array}{rr} \mathit{eval}(p) = n & \text{if } p \longrightarrow^* n^\alpha \\ \mathit{eval}(p) = \textsf{procedure} & \text{if } p \longrightarrow^* \lambda^\alpha[x]t^\alpha \end{array}$$

22.5.1 Redux

Having shown how some of the constructs desugar above, I hope that it is clearer that there are some subtleties involved. The (sapp) rule only makes sense because $[\cdot/\cdot]^\alpha\cdot$ 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 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 ConcreteTerms rather than ConcreteTerms 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 Terms, injections would need to be appropriately placed, and that's it. Even typing contexts Γ could be the same as always.

Similar identification processes can be applied to other things besides bound variable names. For instance, **?** 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 ConcreteTerms using a single ConcreteTerm. Technically when we do so we are almost always referring to the *class* not the individual ConcreteTerm. However we do so using the *notation* of the original ConcreteTerm 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.

[RG: Toggle on and off proof visibility]

22.6 Proofs

Proposition 104 (Principle of Structural Induction for ConcreteTerm)**.** *Let Φ be a property of terms t. Then $\forall t \in \mathit{ConcreteTerm}.\Phi(t)$ if:*

1. *(num)* $\forall n \in \mathbb{N}.\, \Phi(n)$;
2. *(plus)* $\forall t_1, t_2 \in \mathit{ConcreteTerm}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(t_1 + t_2)$;
3. *(var)* $\forall x \in \mathit{Var}.\, \Phi(x)$;
4. *(lam)* $\forall x \in \mathit{Var}, t \in \mathit{ConcreteTerm}.\, \Phi(t) \Rightarrow \Phi(\lambda x.t)$;
5. *(app)* $\forall t_1, t_2 \in \mathit{ConcreteTerm}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(t_1\ t_2)$;
6. *(let)* $\forall x \in \mathit{Var}, t_1, t_2 \in \mathit{ConcreteTerm}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(\mathit{let}\ x = t_1\ \mathit{in}\ t_2)$.

Proposition 105 (Principle of Size Induction for ConcreteTerm)**.**
Let Φ be a property of ConcreteTerms t. Then the following holds:

$$\left(\forall t_1 \in \mathit{ConcreteTerm}.\, \left(\forall t_2 \in \mathit{ConcreteTerm}.\, \mathit{size}(t_2) < \mathit{size}(t_1) \Rightarrow \Phi(t_2)\right) \Rightarrow \Phi(t_1)\right) \Rightarrow \forall t \in \mathit{ConcreteTerm}.\, \Phi(t)$$

Proposition 106 (Principle of Rule Induction for $(\cdot \sim_a \cdot)$)**.** *Let Φ 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. *(αnum)* $\forall n \in \mathbb{N}.\, \Phi(n, n)$;
2. *(αplus)* $\forall (t_{11} \sim_a t_{12}), (t_{21} \sim_a t_{22}).\, \Phi(t_{11}, t_{21}) \wedge \Phi(t_{12}, t_{22}) \Rightarrow \Phi(t_{11} + t_{12}, t_{21} + t_{22})$;
3. *(αvar)* $\forall x \in \mathit{Var}.\, \Phi(x, x)$;
4. *(αlam1)* $\forall x \in \mathit{Var}, (t_1 \sim_a t_2).\, \Phi(t_1, t_2) \Rightarrow \Phi((\lambda x.t_1), (\lambda x.t_2))$;
5. *(αlam2)* $\forall x_1, x_2, x_3 \in \mathit{Var}, t_1, t_2 \in \mathit{ConcreteTerm}.\, x_1 \neq x_2 \wedge x_3 \notin (\mathit{Vars}(t_1) \setminus \{\ \}\, x_1) \cup (\mathit{Vars}(t_2) \setminus \{\ \}\, x_2) \Phi((\lambda x_3.[x_1 \Mapsto x_3]t_1), (\lambda x_3.[x_2 \Mapsto x_3]t_2)) \Rightarrow \Phi((\lambda x_1.t_1), (\lambda x_2.t_2))$;
6. *(αapp)* $\forall (t_{11} \sim_a t_{12}), (t_{21} \sim_a t_{22}).\, \Phi(t_{11}, t_{21}) \wedge \Phi(t_{12}, t_{22}) \Rightarrow \Phi(t_{11}\ t_{12}, t_{21}\ t_{22})$;
7. *(αlet1)*
   $\forall x \in \mathit{Var}, (t_{11} \sim_a t_{12}), (t_{21} \sim_a t_{22}).\, \Phi(t_{11}, t_{21}) \wedge \Phi(t_{12}, t_{22}) \Rightarrow \Phi((\mathit{let}\ x = t_{11}\ \mathit{in}\ t_{12}), (\mathit{let}\ x = t_{11}\ \mathit{in}\ t_{12}))$;
8. *(αlet2)* $\forall x_1, x_2, x_3 \in \mathit{Var}, t_{11}, t_{12} t_{21}, t_{22} \in \mathit{ConcreteTerm}.$
   $x_1 \neq x_2 \wedge x_3 \notin (\mathit{Vars}(t_{12}) \setminus \{\ \}\, x_1) \cup (\mathit{Vars}(t_{22}) \setminus \{\ \}\, x_2) \wedge$
   $\Phi((\mathit{let}\ x_3{=}t_{11}\ \mathit{in}\ [x_1 \Mapsto x_3]t_{12}), (\mathit{let}\ x_3{=}t_{21}\ \mathit{in}\ [x_2 \Mapsto x_3]t_{22})) \Rightarrow \Phi((\mathit{let}\ x_1 = t_{11}\ \mathit{in}\ t_{12}), (\mathit{let}\ x_2 = t_{21}\ \mathit{in}\ t_{22}))$.

Proposition 107 (Principles of Inversion for $(\cdot \sim_a \cdot)$, distinguishing t_1 in $t_1 \sim_a t_2$.)**.**

1. $\forall n \in \mathbb{N}, t_2 \in \mathit{ConcreteTerm}.\, n \sim_a t_2 \Rightarrow t_2 = n$;
2. $\forall t_{11}, t_{12}, t_2 \in \mathit{ConcreteTerm}.\, t_{11} + t_{12} \sim_a t_2 \Rightarrow \exists t_{21}, t_{22} \in \mathit{ConcreteTerm}.\, t_2 = t_{21} + t_{22}$;

3. $\forall x \in \mathit{VAR}, t_2 \in \mathit{CONCRETETERM}.\, x \sim_a t_2 \Rightarrow t_2 = x$;

4.

$$\begin{aligned}\forall x_1 \in \mathit{VAR}, t_{11}, t_2 \in \mathit{CONCRETETERM}.\, (\lambda x_1.t_{11}) \sim_a t_2 \Rightarrow \\ \exists x_2 \in \mathit{VAR}, t_{22} \in \mathit{CONCRETETERM}.\, t_2 = (\lambda x_2.t_{22}) \wedge \\ ((x_2 = x_1 \wedge t_{11} \sim_a t_{22}) \vee \\ (x_2 \neq x_1 \wedge \exists x_3 \in \mathit{VAR}.\, x_3 \notin \mathit{FV}(\lambda x_1.t_{11}) \cup \mathit{FV}(\lambda x_2.t_{22}) \wedge \\ (\lambda x_3.[x_1 \Rightarrow x_3]t_{11}) \sim_a (\lambda x_3.[x_2 \Rightarrow x_3]t_{22})));\end{aligned}$$

5. $\forall t_{11}, t_{12}, t_2 \in \mathit{CONCRETETERM}.\, t_{11}\ t_{12} \sim_a t_2 \Rightarrow \exists t_{21}, t_{22} \in \mathit{CONCRETETERM}.\, t_2 = t_{21}\ t_{22}$;

Proposition 108 (Refined inversion for $(\lambda x_1.t_1) \sim_a (\lambda x_2.t_2)$)**.**

$$\begin{aligned}\forall x_1, x_2 \in \mathit{VAR}, t_1, t_2 \in \mathit{CONCRETETERM}.\, (\lambda x_1.t_1) \sim_a (\lambda x_2.t_2) \Rightarrow \\ \forall x_3 \in \mathit{VAR}.\, x_3 \notin (\mathit{Vars}(t_1) \setminus \{ x_1 \}) \cup (\mathit{Vars}(t_2) \setminus \{ x_2 \}) \Rightarrow \\ (\lambda x_3.[x_1 \Rightarrow x_3]t_{11}) \sim_a (\lambda x_3.[x_2 \Rightarrow x_3]t_{22}).\end{aligned}$$

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. □

Proposition 90 ($(\cdot \sim_a \cdot)$ is an Equivalence Relation)**.**

1. $\forall t \in \mathit{CONCRETETERM}.\, t \sim_a t$. *In words, $\sim_a$ is reflexive;*
2. $\forall t_1, t_2 \in \mathit{CONCRETETERM}.\, t_1 \sim_a t_2 \Rightarrow t_2 \sim_a t_1$. *In words, $\sim_a$ is symmetric;*
3. $\forall t_1, t_2, t_3 \in \mathit{CONCRETETERM}.\, t_1 \sim_a t_2 \wedge t_2 \sim_a t_3 \Rightarrow t_1 \sim_a t_3$. *In words, $\sim_a$ is transitive.*

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

Lemma 11 (num)**.** $\forall n \in \mathbb{N}.\, \Phi(n)$.

Proof. Suppose $n \in \mathbb{N}$. Then by (αnum) $n \sim_a n$. □

Lemma 12 (plus)**.** $\forall t_1, t_2 \in \mathit{CONCRETETERM}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(t_1 + t_2)$.

Proof. Suppose $t_1, t_2 \in$ CONCRETETERM, $t_1 \sim_a t_1$, and $t_2 \sim_a t_2$. Then apply (αplus) to the latter two to yield $t_1 + +t_2 \sim_a t_1 + +t_2$. □

Lemma 13 (var)**.** $\forall x \in \mathit{VAR}.\, \Phi(x)$.

Proof. Suppose $x \in$ VAR. Then by (αvar) $x \sim_a x$. □

Lemma 14 (lam)**.** $\forall x \in \mathit{VAR}, t \in \mathit{CONCRETETERM}.\, \Phi(t) \Rightarrow \Phi(\lambda x.t)$.

Proof. Suppose $x \in$ VAR, $t \in$ CONCRETETERM and $t \sim_a t$. Then apply (αlam) to the latter to yield $(\lambda x.t) \sim_a (\lambda x.t)$. □

Lemma 15 (app)**.** $\forall t_1, t_2 \in \mathit{CONCRETETERM}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(t_1\ t_2)$.

Proof. Suppose $t_1, t_2 \in$ CONCRETETERM, $t_1 \sim_a t_1$, and $t_2 \sim_a t_2$. Then apply (αapp) to the latter two to yield $t_1\ t_2 \sim_a t_1\ t_2$. □

Lemma 16 (let)**.** $\forall x \in \mathit{VAR}, t_1, t_2 \in \mathit{CONCRETETERM}.\, \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(\mathit{let}\ x = t_1\ \mathit{in}\ t_2)$.

Proof. Suppose $x \in$ VAR, $t_1, t_2 \in$ CONCRETETERM, $t_1 \sim_a t_1$, and $t_2 \sim_a t_2$. Then apply (αlet) to the latter two to yield let $x == t_1$ in $t_2 \sim_a$ let $x == t_1$ in t_2. □

□

Proof of 2. Proof by rule induction on $(\cdot \sim_a \cdot)$.

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

Left as an exercise for the reader □

Proof of 3. Proof by size induction on t_2. $\Phi(t_2) \equiv \forall t_1, t_3 \in \textsc{ConcreteTerm}.\, t_1 \sim_a t_2 \wedge t_2 \sim_a t_3 \Rightarrow t_1 \sim_a t_3$. Suppose $t_2 \in \textsc{ConcreteTerm}$ and

$$\Big(\forall t \in \textsc{ConcreteTerm}.\, size(t) < size(t_2) \Rightarrow \Phi(t)\Big) \Rightarrow \Phi(t_2)\Big).$$

Then it suffices to show $\Phi(t_2)$. So suppose $t_1, t_3 \in \textsc{ConcreteTerm}, t_1 \sim_a t_2$, land $t_2 \sim_a t_3$. Then it suffices to show that $t_1 \sim_a t_3$.

We proceed by cases on the structure of $t_2 \in \textsc{ConcreteTerm}$.

Case 43 ($t_2 = n$). [RG: Fill Me In]

Case 44 ($\exists t_{21}, t_{22} \in \textsc{ConcreteTerm}.\, t_2 = t_{21} + t_{22}$). [RG: Fill Me In]

Case 45 ($t_2 = x$). [RG: Fill Me In]

Case 46 ($\exists x \in \textsc{Var}, t_{20} \in \textsc{ConcreteTerm}.\, t_2 = \lambda x.t_{20}$). [RG: Fill Me In]

Case 47 ($\exists t_{21}, t_{22} \in \textsc{ConcreteTerm}.\, t_2 = t_{21}\ t_{22}$). [RG: Fill Me In]

Case 48 ($\exists x \in \textsc{Var}, t_{21}, t_{22} \in \textsc{ConcreteTerm}.\, t_2 = \mathsf{let}\ x = t_{21}\ \mathsf{in}\ t_{22}$). [RG: Fill Me In]

□

Proposition 91 ($[\cdot/\cdot]\cdot$ is well-defined)**.** *The equations in Defn. 20 describe a unique function.*

$$F \in \textsc{ConcreteTerm} \times \textsc{Var} \times \textsc{ConcreteTerm} \to \textsc{ConcreteTerm}.$$

Proof. [RG: FILL ME IN!] □

Proposition 92 (Representation)**.**

1. $\forall t \in \textsc{ConcreteTerm}.\, t \in \mathcal{A}[\![t]\!]$. *In words, each representative is an element of the class it represents;*
2. $\forall t_1, t_2 \in \textsc{ConcreteTerm}.\, t_1 \sim_a t_2 \Leftrightarrow \mathcal{A}[\![t_1]\!] = \mathcal{A}[\![t_2]\!]$.
   In words, $\mathcal{A}[\![\cdot]\!]$ exactly conveys the alpha-equivalence classes;
3. $\forall t^\alpha \in \textsc{Term}. \forall t \in t^\alpha.\, t^\alpha = \mathcal{A}[\![t]\!]$. *In words, any element of t^α can represent it;*
4. $\forall t^\alpha \in \textsc{Term}. \exists t \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t]\!]$. *In words, $\mathcal{A}[\![\cdot]\!]$ is surjective: every Term can be* represented *by a ConcreteTerm.*

Proof.

1. Follows from Prop. 90(1).
2. Let $t_1, t_2 \in \textsc{ConcreteTerm}$. We proceed by cases.

   ($==>$) Suppose $t_1 \sim_a t_2$. Then $t \in \mathcal{A}[\![t_1]\!] \overset{\text{Def. } 22}{\Leftrightarrow} t \sim_a t_1 \overset{\text{Prop. } 90(2,3)}{\Leftrightarrow} t \sim_a t_2 \overset{\text{Def. } 22}{\Leftrightarrow} t \in \mathcal{A}[\![t_2]\!]$.

   ($\Longleftarrow$) Suppose $\mathcal{A}[\![t_1]\!] = \mathcal{A}[\![t_2]\!]$. Then by part (1) $t_1 \in \mathcal{A}[\![t_1]\!]$, $t_2 \in \mathcal{A}[\![t_2]\!]$, and by extensionality, $\forall t \in \textsc{ConcreteTerm}.\, t \in \mathcal{A}[\![t_2]\!] \Leftrightarrow t \in \mathcal{A}[\![t_1]\!]$. Therefore $t_1 \in \mathcal{A}[\![t_2]\!] \overset{\text{Def. } 22}{\Rightarrow} t_1 \sim_a t_2$.
3. Let $t^\alpha \in \textsc{Term}$, and suppose $t \in t^\alpha$. Then for $t_0 \in \textsc{ConcreteTerm}$,

$$t_0 \in t^\alpha \overset{\text{Def. } 21}{\Leftrightarrow} t_0 \sim_a t \overset{\text{Def. } 22}{\Leftrightarrow} t_0 \in \mathcal{A}[\![t]\!].$$

4. Suppose $t^\alpha \in \textsc{Term}$. Then $\exists t \in \textsc{ConcreteTerm}.\, t \in t^\alpha$. Then $t_0 \in t^\alpha \overset{\text{Def. } 21}{\Leftrightarrow} t_0 \sim_a t \overset{\text{Def. } 22}{\Leftrightarrow} t_0 \in \mathcal{A}[\![t]\!]$.

□

Proposition 93 (Liftings)**.**

1. $\forall t_1, t_2 \in \textit{ConcreteTerm}.\, \mathcal{A}[\![t_1]\!] \bullet^\alpha \mathcal{A}[\![t_2]\!] = \mathcal{A}[\![t_1\ t_2]\!]$;
2. $\forall t_1, t_2 \in \textit{ConcreteTerm}.\, \mathcal{A}[\![t_1]\!] +^\alpha \mathcal{A}[\![t_2]\!] = \mathcal{A}[\![t_1 + t_2]\!]$;
3. $\forall x \in \textsc{Var}, t \in \textsc{Term}.\, \lambda^\alpha[x]\mathcal{A}[\![t]\!] = \mathcal{A}[\![\lambda x.t]\!]$.

Proof. [RG: Fill Me In] □

Lemma 17 (Alpha-equivalence preserves size)**.** *If $t_1 \sim_a t_2$ then $size(t_1) = size(t_2)$.*

Proof. Induction on $t_1 \sim_a t_2$. [RG: Fill Me In] □

Lemma 10 (Representational Inversion)**.** *Let $t^\alpha \in \textsc{Term}$. Then exactly one of the following is true:*

1. $\exists! n \in \mathbb{Z}.\, t^\alpha = \mathcal{A}[\![n]\!]$;
2. $\exists! t_1, t_2 \in \textit{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1 + t_2]\!]$;
3. $\exists! x \in \textsc{Var}.\, t^\alpha = \mathcal{A}[\![x]\!]$;
4. $\exists x \in \textsc{Var}, t \in \textit{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\lambda x.t]\!]$;
5. $\exists! t_1, t_2 \in \textit{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1\ t_2]\!]$;
6. $\exists x \in \textsc{Var}, t_1, t_2 \in \textit{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\textit{let}\ x = t_1\ \textit{in}\ t_2]\!]$.

Proof. [RG: Fill Me In] □

Proposition 94 (Structural Inversion)**.** *Let $t^\alpha \in \textsc{Term}$. Then exactly one of the following is true:*

1. $\exists! n \in \mathbb{Z}.\, t^\alpha = n^\alpha$;
2. $\exists! t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, t^\alpha = t_1^\alpha +^\alpha t_2^\alpha$;
3. $\exists! x \in \textsc{Var}.\, t^\alpha = x^\alpha$;
4. $\exists x \in \textsc{Var}, t^\alpha \in \textsc{Term}.\, t^\alpha = \lambda^\alpha[x]t^\alpha$;
5. $\exists! t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, t^\alpha = t_1^\alpha \bullet^\alpha t_2^\alpha$;
6. $\exists x \in \textsc{Var}, t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, t^\alpha = let^\alpha(x, t_1^\alpha, t_2^\alpha)$.

Proof. [RG: Fill Me In] □

Definition 33.

$$size^\alpha : \textsc{Term} \to \mathbb{N}$$
$$size^\alpha(\mathcal{A}[\![t]\!]) = size(t)$$

The definiteness of above definition depends on Prop. 92(4) and Lm. 17.

Lemma 18 (Structural Size Equations)**.**

1. $size^\alpha(n^\alpha) = 1$;
2. $size^\alpha(t_1^\alpha +^\alpha t_2^\alpha) = 1 + size^\alpha(t_1^\alpha) + size^\alpha(t_2^\alpha)$;
3. $size^\alpha(x^\alpha) = 1$;

4. $size^\alpha(\lambda^\alpha[x]t^\alpha) = 1 + size^\alpha(t^\alpha)$;
5. $size^\alpha(t_1^\alpha \bullet^\alpha t_2^\alpha) = 1 + size^\alpha(t_1^\alpha) + size^\alpha(t_2^\alpha)$;
6. $size^\alpha(let^\alpha(x, t_1^\alpha, t_2^\alpha)) = 1 + size^\alpha(t_1^\alpha) + size^\alpha(t_2^\alpha)$;

Proof. [RG: Fill Me In] □

Proposition 96 (Redefinition)**.** $\textsc{Term} \subseteq ST$.

Proof. Suppose $t^\alpha \in \textsc{Term}$. We proceed by induction on $size^\alpha(t^\alpha)$. By Lm. 10, exactly one of the following holds: [RG: Fix to use structural inversion, Prop. ??]

1. $\exists n \in \mathbb{Z}.\, t^\alpha = \mathcal{A}[\![n]\!]$. Then by (STn), $t^\alpha \in \text{ST}$.
2. $\exists t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1 + t_2]\!]$. Then by Prop. 93, $t^\alpha = \mathcal{A}[\![t_1]\!] +^\alpha \mathcal{A}[\![t_2]\!]$, and by Def. 22, $\mathcal{A}[\![t_1]\!], \mathcal{A}[\![t_2]\!] \in \textsc{Term}$. By the induction hypothesis, $\mathcal{A}[\![t_1]\!], \mathcal{A}[\![t_2]\!] \in \text{ST}$ and by (ST+), $t_a \in \text{ST}$.
3. $\exists x \in \textsc{Var}.\, t^\alpha = \mathcal{A}[\![x]\!]$. Then by (STx), $t^\alpha \in \text{ST}$.
4. $\exists x \in \textsc{Var}, t \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\lambda x.t]\!]$;
5. $\exists t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![t_1\ t_2]\!]$;
6. $\exists x \in \textsc{Var}, t_1, t_2 \in \textsc{ConcreteTerm}.\, t^\alpha = \mathcal{A}[\![\text{let } x = t_1 \text{ in } t_2]\!]$.

□

Corollary 2. $\textsc{Term} = ST$.

Proof. Immediate consequence of Def. 24 (of ST) and Prop. 96. □

Proposition 97 (Principle of Rule Induction on the structure of Term)**.** *Let Φ be a predicate on Terms t^α. Then $\Phi(t^\alpha)$ holds for all $t^\alpha \in \textsc{Term}$ if:*

1. $\forall n \in \mathbb{N}.\Phi(n^\alpha)$;
2. $\forall t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha +^\alpha t_2^\alpha)$;
3. $\forall x \in \textsc{Var}.\Phi(x^\alpha)$;
4. $\forall x \in \textsc{Var}, t^\alpha \in \textsc{Term}.\, \Phi(t^\alpha) \Rightarrow \Phi(\lambda^\alpha[x]t^\alpha)$;
5. $\forall t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(t_1^\alpha \bullet^\alpha t_2^\alpha)$;
6. $\forall x \in \textsc{Var}, t_1^\alpha, t_2^\alpha \in \textsc{Term}.\, \Phi(t_1^\alpha) \wedge \Phi(t_2^\alpha) \Rightarrow \Phi(let^\alpha(x, t_1^\alpha, t_2^\alpha))$.

Proof. [RG: Fill Me In] □

Proposition 99 (Lifting)**.** *Let S be some set, and $F : \textsc{ConcreteTerm} \to S$. Then if $\forall t_1, t_2 \in \textsc{ConcreteTerm}.\, t_1 \sim_a t_2 \Rightarrow F(t_1) = F(t_2)$, then there is a unique function $F^\alpha : \textsc{Term} \to S$ such that $\forall t \in \textsc{ConcreteTerm}.\ F^\alpha\ (\mathcal{A}[\![t]\!]) = F(t)$.*

Proof. [RG: Fill Me In] □

Proposition 100. $\forall t_1, t_2 \in \textsc{ConcreteTerm}.\, t_1 \sim_a t_2 \Rightarrow FV(t_1) = FV(t_2)$.

Proof. [RG: Fill Me In] □

Proposition 101. $\forall x \in \textsc{Var}, t^\alpha \in \textsc{Term}.\ FV^\alpha\ (\lambda^\alpha[x]t^\alpha) = FV^\alpha(t^\alpha) \setminus \{\, x \,\}$.

Proof. [RG: Fill Me In] □

Proposition 102 (Capture-Avoiding Substitution Preserves Alpha-equivalence)**.**

$$\forall x \in \mathit{Var}, t_{11}, t_{12}, t_{21}, t_{22} \in \mathit{ConcreteTerm}.\, t_{11} \sim_a t_{21} \wedge t_{12} \sim_a t_{22} \Rightarrow [t_{11}/x]t_{12} \sim_a [t_{21}/x]t_{22}.$$

Proof. [RG: Fill Me In] □

Corollary 3 (Capture-Avoiding Substitution Lifts)**.**
Let $F : \mathit{ConcreteTerm} \times \mathit{Var} \times \mathit{ConcreteTerm} \to \mathit{Term}$ *be defined by* $F = \mathcal{A}[\![\cdot]\!] \circ [\cdot/\cdot]\cdot$*. Then* $\forall x \in \mathit{Var}, t_{11}, t_{12}, t_{21}, t_{22} \in \mathit{ConcreteTerm}.\, t_{11} \sim_a t_{21} \wedge t_{12} \sim_a t_{22} \Rightarrow F(t_{11}, x, t_{12}) = F(t_{21}, x, t_{22})$.

Proof. [RG: Fill Me In] □