New

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853–7501, USA
Email: kozen@cs.cornell.edu

March 17, 2012

Abstract

We propose a theoretical device for modeling the creation of new indiscernible semantic objects during program execution. The method fits well with the semantics of imperative, functional, and object-oriented languages and promotes equational reasoning about higher-order state.

1 Introduction

There are many situations in computing in which we want to create something new. Often we do not really care exactly what is created, as long as it has the right properties. For example, when allocating a new heap cell, we do not care exactly what its address in memory is, but only that we can store and retrieve data there. For that purpose, any heap cell is as good as any other. In object-oriented programming, when we create a new object of a class, we only care that it has the right fields and methods and is different from every other object of that class previously created. In the λ-calculus, when we α-convert to rename a bound variable, we do not care what the new variable is as long as it is fresh.

As common as it is, the intuitive act of creating a new object out of nothing does not fit well with set-theoretic foundations. Such situations are commonly modeled as an allocation of one of a previously existing collection of equivalent candidates. One often sees statements such as, "Let Var be a countable set of variables...," or, "Let $\mathcal{L}$ be a countable set of heap cells..." The set is assumed to exist in advance of its use and is assumed to be large enough that fresh elements can always be obtained when needed. Standard references on the semantics of objects also tend to treat object creation as allocation [1, 17, 19].

The difficulty here is that the candidates for allocation should be theoretically indiscernible, whereas real implementations must somehow make a de-

terministic choice. But to choose requires some way of distinguishing the chosen from the unchosen, thus the candidates cannot be indiscernible after all. Moreover, cardinality constraints often interfere with closure conditions on the language. For example, we only need a countable set of variables to represent an infinitary λ-term, but if all available variables already occur in the term, there would be none left over in case we needed a fresh one for α-conversion. One could permute the variables to free one up, but that is awkward.

The issue is related to the philosophical problem of the *identity of indiscernibles*. Leibniz proposed that objects that have all the same properties must in fact be the same object. Although the subject of much debate in the philosophical literature [6, 7, 10], it is certainly desirable in programming language semantics, especially object-oriented programming, to allow the existence of distinct but indiscernible semantic objects. But it can also be the source of much confusion, as is well known to anyone who has ever tried to explain to introductory Java students why one should never compare strings with `==`.

The issue also arises in systems involving terms with variable binders, such as quantificational logic and the λ-calculus. We would like to treat bound variables as indiscernible for the purposes of α-conversion and safe (capture-avoiding) substitution. Several devices for the generation of fresh variables have been proposed, both practical and theoretical, the earliest possibly being the gensym facility of LISP. Popular variable-avoiding alternative representations of λ-terms include de Bruijn indices and Stoy diagrams [4]. The NuPrl system [2, 5] has a facility for generating *nonces*, or objects for which nothing can be tested except identity. A particularly appealing approach for reasoning about syntactic terms with binders is *nominal logic* [8, 9, 20, 21]. Notions of equivalence under α-conversion and freeness are handled quite elegantly in this framework. Structures are assumed to be invariant with respect to permutations on the candidates.

In this paper we propose a device for creating new indiscernible objects in a semantic domain. Simply put, a semantic object is created by allocating a name for it. The object itself is *defined to be* the congruence class of all its names. A system such as nominal logic can be used to handle the generation of names in the syntactic domain.

The idea can be illustrated with a very simple example. Consider a domain of semantic objects $D = \{a, b, c, \ldots\}$. Let φ be a first-order formula with free variables, say $x = y \wedge y \neq z$. According to the usual Tarskian definition of truth, we could interpret φ relative to a valuation $\sigma : \mathsf{Var} \to D$, provided $\{x, y, z\} \subseteq \operatorname{dom} \sigma$, and the judgment $\sigma \models \varphi$ would have a well-defined truth value. For example, if $\sigma(x) = \sigma(y) = c$ and $\sigma(z) = a$, then σ would satisfy φ, along with many other other valuations over D.

However, suppose we did not specify the actual values of x, y, z, but only which variables represent the *same* values. Thus instead of $\sigma : \mathsf{Var} \to D$, we would have a set of equations $\alpha \subseteq \mathsf{Var} \times \mathsf{Var}$ specifying aliasing relationships between the variables. For the σ above, α would consist of the single equation

$x = y$. The free algebra generated by $\{x, y, z\}$ modulo the congruence induced by $x = y$ has two elements, namely the two congruence classes $\{x, y\}$ and $\{z\}$. Under the canonical interpretation

$$x \mapsto \{x, y\} \qquad y \mapsto \{x, y\} \qquad z \mapsto \{z\},$$

the formula φ is satisfied. The presentation α of the free algebra contains enough information to determine the truth of the formula; there is no need to represent the actual values.

The relation α is called an *aliasing relation*. It generates a congruence, that is, the smallest relation on terms that contains α and is reflexive, symmetric, transitive, and a congruence with respect to any operations defined on the elements. To represent the creation of a new object, we simply update α in a way that ensures that there is no aliasing between the variable instantiated with the new object and others of the same type currently represented in the state. We do not have to worry about how to select a new semantic object from a previously defined set or whether there are enough of them available; in essence, that responsibility is completely borne by the allocation of syntactic names. The advantage of this approach is that objects in the semantic domain can be generated *ex nihilo* and are truly indiscernible. An added benefit is that we can reason equationally about the program state using α.

In this paper we develop this basic idea into an operational semantics for a higher-order programming language with functional, imperative, and object-oriented features. We give a set of operational rules that describe how the state, as represented by σ and α, should be updated as each atomic action is performed. The semantics is an extension of *capsules* [12, 13, 14]. We show how objects, nonces, references, arrays, and records fit into this framework. As an illustration, we show how to model safe substitution in the λ-calculus with nonces as variables and show that α-conversion is an idempotent operation.

2 Background

2.1 Capsules

Capsules [12, 13, 14] are a precursor to the system introduced here. Capsule semantics does not rely on heaps, stacks, or any other form of explicit memory, but only on names and bindings.

2.1.1 Syntax

A *capsule* is a pair $\langle e, \sigma \rangle$, where e is a λ-term or constant and σ is a partial function from variables to irreducible λ-terms or constants such that

(i) $\mathsf{FV}(e) \subseteq \text{dom } \sigma$, and

(ii) if $x \in \text{dom}\ \sigma$, then $\mathsf{FV}(\sigma(x)) \subseteq \text{dom}\ \sigma$,

where $\mathsf{FV}(e)$ is the set of free variables of e. Thus the free variables of a capsule are not really free; every variable in $\langle e, \sigma \rangle$ either occurs in the scope of a λ or is bound by σ to a constant or irreducible expression, which represents its value. A capsule represents a closed λ-coterm (infinitary λ-term). The closure conditions (i) and (ii) preclude catastrophic failure due to access of unbound variables. There may be circularities, which enables a representation of recursive functions.

Capsules may be α-converted. Abstraction operators λx and the occurrences of x bound to them may be renamed as usual. Variables in dom σ may also be renamed along with all free occurrences. Capsules that are equivalent in this sense represent the same value.

Values are also preserved by *garbage collection*. A *monomorphism of capsules* $h : \langle d, \sigma \rangle \to \langle e, \tau \rangle$ is an injective map $h : \text{dom}\ \sigma \to \text{dom}\ \tau$ such that

- $\tau(h(x)) = h(\sigma(x))$ for all $x \in \text{dom}\ \sigma$, and
- $h(d) = e$,

where $h(e) = e[x/h(x)]$ (safe substitution). The set of monomorphic preimages of a given capsule contains an initial object that is unique up to a permutation of variables. This is the garbage-collected version of the capsule.

2.1.2 Semantics

Capsule evaluation semantics looks very much like the original evaluation semantics of LISP, with the added twist that a fresh variable is substituted for the parameter in function applications. The relevant small-step rule is

$$\langle (\lambda x.e)\ v, \sigma \rangle \to \langle e[x/y], \sigma[y/v] \rangle,$$

where y is fresh. In the original evaluation semantics of LISP, the right-hand side is $\langle e, \sigma[x/v] \rangle$, which gives dynamic scoping. This simple change faithfully models β-reduction with safe substitution in the λ-calculus, providing static scoping without closures [12, 13]. It also handles local variable declaration in recursive functions correctly.

Another evaluation rule of particular note is the assignment rule:

$$\langle x := v, \sigma \rangle \to \langle (), \sigma[x/v] \rangle$$

where v is irreducible. The closure condition (i) of §2.1.1 ensures that x is already bound in σ, and the assignment rebinds x to v. Assignment is also used to create recursive functions via backpatching, also known as *Landin's knot*, without the use of fixpoint combinators.

See [12, 13, 14] for further details and examples.

2.2 Nominal Logic

Nominal logic [8, 9, 20, 21] can be used to handle the allocation of fresh program variables. We review the basic definitions of nominal logic here.

Let N be a countably infinite set of *names* and let S_N be the group of permutations on N. A *group action* of S_N on a set U is a group homomorphism $\tau : S_N \to S_U$. Given a group action τ, we say that $u \in U$ is *supported by* $S \subseteq N$ if $\tau(g)$ fixes u whenever $g \in S_N$ fixes S pointwise. An element of U is *finitely supported* if it is supported by some finite subset of N.

A *nominal set* is a set U equipped with a group action $\tau : S_N \to S_U$ such that all elements of U are finitely supported.

The set of subsets of N supporting U is closed under superset and intersection. Thus if u has finite support, then it has a least finite support, denoted $\mathsf{supp}\, u$. If $x \in N - \mathsf{supp}\, u$, one says that x is *fresh for* u and write $x\#u$.

The canonical example of a nominal set is the set Λ of finite λ-terms over λ-variables N. Let $\tau : S_N \to S_\Lambda$ take $g : N \to N$ to the action $\tau(g) : \Lambda \to \Lambda$ that substitutes $g(x)$ for x all occurrences of all variables x in a term to $g(x)$. The support of a term is the set of variables occurring in the term.

3 Syntax

In this section we define the syntax of our language. We use the same notation for rebinding and substitution. Given a function σ, we write $\sigma[x/v]$ for the function such that $\sigma[x/v](y) = \sigma(y)$ for $y \neq x$ and $\sigma[x/v](x) = v$. Given an expression e, we write $e[x/d]$ for the expression e with d substituted for all free occurrences of x, renaming bound variables as necessary to avoid capture.

3.1 Types

Our type system distinguishes between *constructive types* and *creative types*. Constructive objects are constructed from other objects (and perhaps themselves, in the case of coinductive types) using constructors. They are represented directly in the state, bound by an environment σ to a variable of the same type. Creative objects, on the other hand, do not exist in advance and are not built from constructors, but are created on the fly during program execution using **new**. They can be used to model objects (in the sense of object-oriented programming), references, arrays, records, and nonces. Creative objects have a weaker ontological status than constructive objects in that they have no direct representation in the state, but only indirect representation in the form of an aliasing relation α.

The collection of all types is denoted Type. Let $\mathsf{Var} = \{x, y, z, \ldots\}$ be an unlimited supply of *variables*. A *type environment* is a partial function $\Gamma : \mathsf{Var} \rightharpoonup$

Type with finite domain dom Γ.

3.1.1 Constructive Types

Constructive types are built from type constructors. We have the function space constructor $\to$, products and coproducts, and coinductive types defined with finite systems of fixpoint equations.

Products and coproducts are of the form

$$\prod \Gamma = \prod_{x \in \mathrm{dom}\ \Gamma} \Gamma(x) \qquad\qquad \sum \Gamma = \sum_{x \in \mathrm{dom}\ \Gamma} \Gamma(x)$$

where Γ is a type environment. The corresponding projections and injections have type

$$\pi_x : \prod \Gamma \to \Gamma(x) \qquad\qquad \iota_x : \Gamma(x) \to \sum \Gamma$$

for $x \in \mathrm{dom}\ \Gamma$. The unit type $\mathbb{1}$ is the empty product, and the type of booleans is $\mathbb{2} = \mathbb{1} + \mathbb{1}$.

Our product and coproduct types are not dependent types, as Var is not a type. All function, product, and coproduct types are constructive.

3.1.2 Creative Types

In addition to constructive types, we have *creative types* $\mathcal{C}(\prod \Gamma)$, where $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{Type}$ is a type environment. The type $\mathcal{C}(\prod \Gamma)$ represents a class of objects having fields named x for $x \in \mathrm{dom}\ \Gamma$ in the sense of object-oriented programming. Values of type $\mathcal{C}(\prod \Gamma)$ are creative objects. The field x has type $\Gamma(x)$, which can be either constructive or creative.

3.1.3 Coinductive Types

Coinductive types are defined by finite systems of fixpoint equations. For example, the natural numbers $\mathbb{N}$ are the type $\mathbb{N} = \mathbb{1} + \mathbb{N}$. Integer lists and streams are defined by

$$\mathsf{intlist} = \mathbb{1} + (\mathbb{N} \times \mathsf{intlist})$$

where $\mathbb{N}$ and intlist are type variables. Formally, $\mathsf{intlist} = \sum \Delta$, where

$$\Delta(\mathsf{nil}) = \mathbb{1} \qquad \Delta(\mathsf{cons}) = \prod \Gamma \qquad \Gamma(\mathsf{hd}) = \mathbb{N} \qquad \Gamma(\mathsf{tl}) = \mathsf{intlist}$$

Then

$$\begin{array}{lll} \sum \Delta = \mathbb{1} + \prod \Gamma & \iota_{\mathsf{nil}} : \mathbb{1} \to \sum \Delta & \iota_{\mathsf{cons}} : \prod \Gamma \to \sum \Delta \\ \prod \Gamma = \mathbb{N} \times \mathsf{intlist} & \pi_{\mathsf{hd}} : \prod \Gamma \to \mathbb{N} & \pi_{\mathsf{tl}} : \prod \Gamma \to \mathsf{intlist} \end{array}$$

Both constructive and creative types may appear in a coinductive type definition.

3.2 Expressions

Expressions $d, e, \ldots$ are defined inductively. Variables are expressions, as are typed projections π_x and injections ι_x. The unit object $()$ is the null tuple of type $\mathbb{1}$, and booleans $0 = \iota_{\mathsf{false}}()$ and $1 = \iota_{\mathsf{true}}()$ are of type $\mathbb{2}$.

Compound expressions are formed with the following constructs, subject to typing constraints.

- λ-abstraction $\quad \lambda x.e$
- application $\quad (d\ e)$
- assignment $\quad x := e$ or $d.x := e$
- tupling $\quad (e_x \mid x \in \mathrm{dom}\ \Gamma)$
- case analysis $\quad [e_x \mid x \in \mathrm{dom}\ \Gamma]$
- projection $\quad e.x$
- object creation $\quad \mathsf{new}\ \Gamma(e)$
- identity test $\quad d = e$

We also have defined expressions

- composition $\quad d\,;\,e \quad (\lambda x.e)\,d$
- conditional $\quad \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e \quad [\lambda x.d,\ \lambda x.e]\ b$
- while loop $\quad \mathsf{while}\ b\ \mathsf{do}\ e \quad \mathsf{let\ rec}\ w = \lambda x.\mathsf{if}\ b\ \mathsf{then}\ e\,;\,w\ ()\ \mathsf{else}\ ()\ \mathsf{in}\ w\ ()$
- local definition $\quad \mathsf{let}\ x = d\ \mathsf{in}\ e \quad (\lambda x.e)\,d$
- recursive definition $\quad \mathsf{let\ rec}\ x = d\ \mathsf{in}\ e \quad \mathsf{let}\ x = \bot\ \mathsf{in}\ (x := d)\,;\,e$

It is not necessary to worry about the capture of free occurrences of x in the composition, conditional, and while loop because the type of x is $\mathbb{1}$ in all cases.

The $\bot$ in the definition of let rec is a special constant of the appropriate type designated for this purpose. This technique is known as *Landin's knot*. The constant $\bot$ is treated specially in the small-step operational semantics (see §4.1) in that a variable bound to it is considered irreducible, effectively allowing Landin's knot to create self-referential objects.

The let rec construct is used to create recursive functions and values of coinductive types. It specifies a value that is the unique solution of the given equation in a final coalgebra. For recursive functions, this is a λ-coterm (infinitary λ-term), as in capsules (see §2.1). For coinductive datatypes, it is a realization

of the type as defined in [16]. In both cases, the infinite object is regular and has a finite representation. For example, the type of integer lists and streams was defined in §3.1.3. An element of this type is the infinite stream of alternating 0s and 1s, which can be defined by

$$\mathsf{let\,rec}\ x = \iota_{\mathsf{cons}}(0, \iota_{\mathsf{cons}}(1, x))\ \mathsf{in}\ e$$

The mutually recursive definition

$$\mathsf{let\,rec}\ x_1 = d_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_n = d_n\ \mathsf{in}\ e$$

can be coded using the single-variable form of let rec with products and projections or with nested let recs.

The case analysis construct $[e_x \mid x \in \mathrm{dom}\ \Gamma]$ corresponds to a case or match statement of functional languages. It is used to extract the elements of a coproduct based on their types. For example, the map function that maps a given function $f : \mathbb{N} \to \mathbb{N}$ over a given integer list would be defined by let rec to satisfy the equation

$$\mathsf{map} = \lambda(f : \mathbb{N} \to \mathbb{N}).[\iota_{\mathsf{nil}},\ \lambda x.\iota_{\mathsf{cons}}(f(\pi_{\mathsf{hd}}\, x),\ \mathsf{map}\ f\ (\pi_{\mathsf{tl}}\, x))]$$

This would be written more conventionally as

$$\begin{array}{l}\mathsf{map}\,(f : \mathbb{N} \to \mathbb{N})\,(\ell : \mathsf{intlist}) : \mathsf{intlist} = \\ \quad \mathsf{case}\ \ell\ \mathsf{of} \\ \quad\quad \mid \iota_{\mathsf{nil}}() \to \iota_{\mathsf{nil}}() \\ \quad\quad \mid \iota_{\mathsf{cons}}\, x \to \iota_{\mathsf{cons}}(f(\pi_{\mathsf{hd}}\, x),\ \mathsf{map}\ f\ (\pi_{\mathsf{tl}}\, x))\end{array}$$

3.2.1 Typing Rules

Let $\Delta : \mathsf{Var} \rightharpoonup \mathsf{Type}$ be a type environment. We write $\Delta \vdash e : \alpha$ if the type α can be derived for the expression e by the typing rules of Fig. 1. The constructs let rec $x = d$ in e and let $x = d$ in e are typed as $(\lambda x.e)\, d$.

If $\Delta \vdash e : \alpha$ for some α, we say that e is Δ-*well-typed*, or just *well-typed* if Δ is understood. Unless otherwise mentioned, we will assume that the use of an expression in the text implies that it is well-typed.

3.2.2 Assignable Expressions

An *assignable expression* is a Δ-well-typed expression of the form $x_0.x_1.\ldots.x_n$, $n \geq 0$, where $x_i \in \mathsf{Var}$. It follows from the typing rules that each nonull proper prefix is creative; that is, there are Γ_i for $0 \leq i \leq n$ such that $\Delta = \Gamma_0$, $\Gamma_i(x_i) = \mathcal{C}(\prod \Gamma_{i+1})$ for $0 \leq i \leq n-1$, and $x_n \in \mathrm{dom}\ \Gamma_n$. An assignable expression may appear on the left-hand side of an assignment operator := and is considered irreducible when appearing in that position (although non-irreducible

$$\Delta \vdash c : \mathsf{type}(c),\ c \in \mathsf{Const} \qquad \Delta \vdash x : \Delta(x),\ x \in \mathrm{dom}\ \Delta$$

$$\frac{\Delta \vdash x : \alpha \quad \Delta \vdash e : \beta}{\Delta \vdash \lambda x.e : \alpha \to \beta} \qquad \frac{\Delta \vdash d : \alpha \to \beta \quad \Delta \vdash e : \alpha}{\Delta \vdash (d\ e) : \beta}$$

$$\frac{\Delta \vdash x : \alpha \quad \Delta \vdash e : \alpha}{\Delta \vdash x := e : \mathbb{1}} \qquad \frac{\Delta \vdash d.x : \alpha \quad \Delta \vdash e : \alpha}{\Delta \vdash d.x := e : \mathbb{1}}$$

$$\frac{\Delta \vdash b : \mathbb{2} \quad \Delta \vdash d : \alpha \quad \Delta \vdash e : \alpha}{\Delta \vdash \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e : \alpha} \qquad \frac{\Delta \vdash b : \mathbb{2} \quad \Delta \vdash e : \mathbb{1}}{\Delta \vdash \mathsf{while}\ b\ \mathsf{do}\ e : \mathbb{1}}$$

$$\frac{\Delta \vdash d : \mathbb{1} \quad \Delta \vdash e : \alpha}{\Delta \vdash d\,;\, e : \alpha} \qquad \frac{\Delta \vdash d : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash e : \mathcal{C}(\prod \Gamma)}{\Delta \vdash d = e : \mathbb{2}}$$

$$\frac{\Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Gamma \vdash x : \beta}{\Delta \vdash e.x : \beta} \qquad \frac{\Delta \vdash e : \prod \Gamma}{\Delta \vdash \mathsf{new}\ \Gamma(e) : \mathcal{C}(\prod \Gamma)}$$

$$\frac{\Delta \vdash e_x : \Gamma(x),\ x \in \mathrm{dom}\ \Gamma}{\Delta \vdash (e_x \mid x \in \mathrm{dom}\ \Gamma) : \prod \Gamma} \qquad \frac{\Delta \vdash e_x : \Gamma(x) \to \beta,\ x \in \mathrm{dom}\ \Gamma}{\Delta \vdash [e_x \mid x \in \mathrm{dom}\ \Gamma] : \sum \Gamma \to \beta}$$

Figure 1: Typing Rules

expressions may appear on the left-hand side of an assignment). The set of Δ-well-typed assignable expressions is denoted AExp_Δ, or just AExp if Δ is understood. This set can be infinite in general due to coinductive types, but it is a regular set considered as a set of strings over Var. Assignable expressions are denoted $u, v, w, \ldots$.

Assignable expressions can be either constructive or creative. The set of Δ-well-typed creative (respectively, constructive) assignable expressions is denoted CExp_Δ (respectively, NExp_Δ), or just CExp (respectively, NExp) if Δ is understood. Like AExp, these sets can be infinite in general.

3.3 Aliasing Relations

Let $\alpha \subseteq \mathsf{CExp} \times \mathsf{CExp}$ be a set of pairs of creative assignable expressions such that if $(u, v) \in \alpha$, then $\Delta \vdash u : \mathcal{C}(\prod \Gamma)$ iff $\Delta \vdash v : \mathcal{C}(\prod \Gamma)$. The set α is called an *aliasing relation*. It represents a set of well-typed equations between creative assignable expressions.

The *congruence generated by* α is the smallest binary relation on AExp_Δ containing α and closed under the rules of Fig. 2. There is some redundancy among

$$\frac{(u,v) \in \alpha}{\alpha \vdash u = v} \qquad \frac{\Delta \vdash u : \beta}{\alpha \vdash u = u} \qquad \frac{\alpha \vdash u = v}{\alpha \vdash v = u} \qquad \frac{\alpha \vdash u = v \quad \alpha \vdash v = w}{\alpha \vdash u = w}$$

$$\frac{\Delta \vdash u : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash v : \mathcal{C}(\prod \Gamma) \quad x \in \operatorname{dom} \Gamma \quad \alpha \vdash u = v}{\alpha \vdash u.x = v.x}$$

Figure 2: Congruence Rules

the premises of the last rule (congruence), as one can show inductively that if $\alpha \vdash u = v$, then u and v have the same type. Note that u and v can be constructive, even though the elements of α are all creative. The congruence class of $v \in \mathsf{AExp}$ is denoted $[v]_\alpha$.

We can form the free algebra $\mathsf{AExp}_\Delta/\alpha = \{[u]_\alpha \mid u \in \mathsf{AExp}_\Delta\}$. It is an algebra in the sense that the projections $.x$, regarded as unary operations, are well-defined on congruence classes; that is, if $[u]_\alpha = [v]_\alpha$, then by congruence, $[u.x]_\alpha = [v.x]_\alpha$ whenever $u.x$ is well-typed, so it makes sense to define $[u]_\alpha.x = [u.x]_\alpha$. Intuitively, if $\Delta \vdash u = v$, then u and v are aliases for the same object, so the values of the fields $u.x$ and $v.x$ should also be the same.

As mentioned, the set AExp can be infinite in general, but the computational rules will maintain the invariant that AExp/α is finite. One can regard AExp/α as a finite graph with nodes $[u]_\alpha$ and labeled edges $[u]_\alpha \xrightarrow{x} [u.x]_\alpha$.

We denote by $\mathsf{CExp}_\Delta/\alpha$ and $\mathsf{NExp}_\Delta/\alpha$ the sets of creative and constructive elements of $\mathsf{AExp}_\Delta/\alpha$, respectively; that is, the sets

$$\mathsf{CExp}_\Delta/\alpha = \{[u]_\alpha \mid u \in \mathsf{CExp}_\Delta\}$$
$$\mathsf{NExp}_\Delta/\alpha = \{[u]_\alpha \mid u \in \mathsf{NExp}_\Delta\} = \mathsf{AExp}_\Delta/\alpha - \mathsf{CExp}_\Delta/\alpha.$$

3.4 Equational Reasoning

The congruence generated by α extends inductively to effect-free constructors with the obvious syntactic congruence rule for each constructor. For example, for products and injections,

$$\frac{\alpha \vdash d_x = e_x,\ x \in \operatorname{dom} \Gamma}{\alpha \vdash (d_x \mid x \in \operatorname{dom} \Gamma) = (e_x \mid x \in \operatorname{dom} \Gamma)} \qquad \frac{\alpha \vdash d = e}{\alpha \vdash \iota_x\, d = \iota_x\, e}.$$

The only nonobvious rule is λ-abstraction, in which we must treat the bound variable specially.

$$\frac{\alpha \vdash d[x/z] = e[x/z],\ z \text{ fresh}}{\alpha \vdash \lambda x.d = \lambda x.e}.$$

There is no sound congruence rule for assignment $:=$ or new, as these constructs have side effects.

These rules, along with α-conversion, renaming by a permutation, garbage collection (§3.6), and reduction will enable equational reasoning on program states.

3.5 Program States

A program state is represented by a quadruple $\langle e, \Delta, \sigma, \alpha \rangle$, where:

- $\Delta : \mathsf{Var} \rightharpoonup \mathsf{Type}$ is a type environment
- $\alpha \subseteq \mathsf{CExp}_\Delta \times \mathsf{CExp}_\Delta$ is a Δ-well-typed equational presentation
- $\sigma : \mathsf{NExp}_\Delta / \alpha \to \mathsf{NExp}_\Delta$ is a Δ-well-typed *valuation*
- e is a Δ-well-typed expression

such that if $\Delta \vdash u : \beta$ and β is constructive, then $[u]_\alpha \in \operatorname{dom} \sigma$. The domain of σ is officially $\mathsf{NExp}_\Delta / \alpha$, but we will often abuse notation and write $\sigma(u)$ for $\sigma([u]_\alpha)$.

The first component e is the expression to be evaluated. The typing of expressions is determined by Δ. The components σ and α comprise an environment that determines the interpretation of free variables. The condition on the domain of σ takes the place of condition (ii) of §2.1.1 for capsules, and condition (i) is implied by the fact that e is well-typed.

The set of states is a nominal set over the set of names Var (§2.2).

3.6 Garbage Collection

Our notions of α-conversion and garbage collection are based on capsules (see §2.1.1) with appropriate modifications to account for the aliasing relation α. As with capsules, values are preserved. These are important aspects of our language, as they allow equational reasoning.

Any variable declared in Δ may be α-converted. If a fresh variable is needed for α-conversion, its type is first declared in Δ. Renaming variables in some type environment Γ used in the declaration of a product or sum does not constitute α-conversion and does not result in an equivalent state.

As with capsules, garbage collection is defined in terms of monomorphisms. A *monomorphism*

$$h : \langle e, \Delta, \sigma, \alpha \rangle \to \langle e', \Delta', \sigma', \alpha' \rangle$$

is an injective map $h : \operatorname{dom} \Delta \to \operatorname{dom} \Delta'$ such that

(i) h is type-preserving, that is, $\Delta(x) = \Delta'(h(x))$;

(ii) modulo α and α', h is an algebra monomorphism $\mathsf{AExp}_\Delta/\alpha \to \mathsf{AExp}_{\Delta'}/\alpha'$;

(iii) $\sigma'([h(x)]_{\alpha'}) = h(\sigma([x]_\alpha))$ for all $[x]_\alpha \in \mathrm{dom}\ \sigma$; and

(iv) $e' = h(e)$,

where $h(e) = e[x/h(x)]$. Like capsules, every state has an initial monomorphic preimage, which is its garbage-collected version and which is unique up to a permutation of variables and variation in the presentation α of $\mathsf{AExp}_\Delta/\alpha$.

However, unlike capsules, we cannot collect garbage simply by removing variables inaccessible from e, because some of them may be needed in the equational presentation α of $\mathsf{AExp}_\Delta/\alpha$. Removing the equations containing them could cause property (ii) to be violated; h would be a homomorphism but not a monomorphism. To ensure (ii), we show that $\mathsf{AExp}_\Delta/\alpha$ has a canonical presentation in which α is minimal and the pairs are of a certain form.

Lemma 3.1 *Given an aliasing relation α on AExp_Δ, there is a set of variables X, an extension Δ' of Δ with domain $X \cup \mathrm{dom}\ \Delta$, and an aliasing relation α' on $\mathsf{AExp}_{\Delta'}$ with the following properties:*

(i) *$\mathsf{AExp}_\Delta/\alpha$ and $\mathsf{AExp}_{\Delta'}/\alpha'$ are isomorphic;*

(ii) *all pairs in α' are of the form (x, z) or $(x.y, z)$, where $x, z \in X$;*

(iii) *every congruence class in $\mathsf{CExp}_{\Delta'}/\alpha'$ contains exactly one variable of X.*

Moreover, Δ' and α' can be computed from Δ and α in almost linear time ($O(n\,\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function).

Proof. Let A be the set of subterms of terms appearing in α. Form the congruence closure $\widehat{\alpha}$ of α on A. The congruence closure is the smallest relation on A that contains α and is closed under the rules of Fig. 2 applied only to terms in A. It is shown in [15] that for $s, t \in A$, $\alpha \vdash s = t$ iff $(s, t) \in \widehat{\alpha}$; that is, one need not go outside of A to prove congruence between two terms in A.

It is known how to form the congruence closure for a signature involving only unary functions in almost linear time [3, 11, 18]. By "forming the congruence closure," we do not mean computing the relation $\widehat{\alpha}$ itself—that would take too long to write down—but rather forming the congruence classes and pointers from elements of A to their respective congruence classes so that we can subsequently determine whether $(s, t) \in \widehat{\alpha}$ (that is, $\alpha \vdash s = t$) for $s, t \in A$ in constant time.

Let X be a set of variables such that each creative α-congruence class contains exactly one element of X. If $[u]_\alpha$ does not contain a variable, we can add a fresh variable x and the equation (x, u) to α, although this step is not strictly

necessary, as our operational semantics maintains the invariant that every creative congruence class contains a variable. Let Δ' be Δ extended as necessary with the appropriate typings for $x \in X$.

Now let

$$\alpha' = \widehat{\alpha} \cap (\{(x, z) \mid x \in X,\ z \in \mathsf{Var}\} \cup \{(x.y, z) \mid x, z \in X\}).$$

The set α' has the following properties:

- For each $u \in A$, there is exactly one $x \in X$ such that $\alpha' \vdash x = u$.
- For each $x \in X$ and $y \in \operatorname{dom} \Gamma$, where $\Delta'(x) = \mathcal{C}(\prod \Gamma)$, there is exactly one $z \in X$ such that $(x.y, z) \in \alpha'$.

It follows that α and α' generate the same congruence closure $\widehat{\alpha}$, thus $\mathsf{AExp}_{\Delta}/\alpha$ and $\mathsf{AExp}_{\Delta'}/\alpha'$ are isomorphic. □

Now we can collect garbage by forming the reduced presentation as described in Lemma 3.1 and removing inaccessible variables from Δ, σ, and α, where a variable is *accessible* if it is in the smallest set of variables containing the variables of e and closed under the following operations:

- If x is accessible, $(x, z) \in \alpha$ or $(x.y, z) \in \alpha$, and $z \in X$, then z is accessible;
- if x is accessible and z occurs in $\sigma([x]_\alpha)$ or $\sigma([x.y]_\alpha)$, then z is accessible.

The image of the monomorphism h is the subalgebra of $\mathsf{AExp}_{\Delta}/\alpha$ generated by the accessible variables.

4 Operational Semantics

The operational semantics of the language is defined by the small-step rules given below. In addition, there are context rules that define a standard shallow applicative-order evaluation strategy (leftmost innermost, call-by-value) and left-to-right evaluation of tuples and expressions $e.x$.

4.0.1 Irreducible States

Irreducible states are those for which no small-step operational rule applies. Mostly (but not always), irreducible states are defined by their first component, the expression to be reduced. The state $\langle e, \Delta, \sigma, \alpha \rangle$ is irreducible if e is

- a constant,
- a λ-abstraction,

- a creative assignable expression, i.e. an element of CExp_Δ,
- a variable x such that $\sigma(x) = \bot$,
- an expression $(v_x \mid x \in \operatorname{dom}\Gamma)$, where all v_x are irreducible,
- an expression $[v_x \mid x \in \operatorname{dom}\Gamma]$, where all v_x are irreducible.
- an expression $\iota_x(v)$, where v is irreducible.

Note that assignable expressions of constructive type are not irreducible.

4.1 Operational Rules

4.1.1 Function Application

Our rule for function application is adapted from the rule for capsules (see §2.1.2):

$$\langle (\lambda x.e)\ v,\ \Delta,\ \sigma,\ \alpha \rangle \ \rightarrow\ \langle e[x/y],\ \Delta[y/\Delta(x)],\ \sigma',\ \alpha' \rangle,$$

where y is fresh and

$$(\sigma',\ \alpha') = \begin{cases} (\sigma[[y]_\alpha / v],\ \alpha) & \text{if } \Delta(x) \text{ is constructive} \\ (\sigma,\ \alpha \cup \{(y, v)\}) & \text{if } \Delta(x) \text{ is creative.} \end{cases}$$

As with capsules, a fresh variable y is conjured and given the same type as x, resulting in a new global type environment $\Delta[y/\Delta(x)]$. If the type is constructive, σ is updated with the value v, and α is unchanged. If the type is creative, σ is unchanged, but α is updated with the new alias (y, v).

4.1.2 Creation

The following rule creates a new creative object:

$$\langle \mathsf{new}\ \Gamma(v),\ \Delta,\ \sigma,\ \alpha \rangle \ \rightarrow\ \langle y,\ \Delta[y/\mathcal{C}(\textstyle\prod \Gamma)],\ \sigma',\ \alpha' \rangle,$$

where y is fresh and

$$\alpha' = \alpha \cup \{(y.x,\ v_x) \mid x \in \operatorname{dom}\Gamma,\ \Gamma(x) \text{ creative}\}$$
$$\sigma' = \sigma[[y.x]_{\alpha'} / v_x \mid x \in \operatorname{dom}\Gamma,\ \Gamma(x) \text{ constructive}]$$

The object is represented by a fresh variable y, which is added to the domain of Δ with the appropriate creative type. The value v is a tuple supplying the initial values of the fields. The entities α and σ are updated to assign the fields of the new object their initial values.

4.1.3 Assignment to Constructive Expressions

Assignment for constructive types is essentially the same as for capsules. For $u \in \mathsf{NExp}$ and v irreducible of the same constructive type,

$$\langle u := v,\, \Delta,\, \sigma,\, \alpha \rangle \;\to\; \langle (),\, \Delta,\, \sigma[[u]_\alpha / v],\, \alpha \rangle.$$

Here Δ does not need to be updated, because u is already well-typed.

4.1.4 Assignment to Creative Variables

Before we can define the semantics of assignments to creative assignable expressions, we need to lay some groundwork. The issue is that assignment to a creative expression may change the free algebra presented by α if the expression to be assigned is involved in the presentation.

First we consider the case of an assignment $x := v$ to a creative variable $x \in \operatorname{dom} \Delta$. Let $\Delta' = \Delta[z/\Delta(x)]$, where $z \notin \operatorname{dom} \Delta$. Define $g : \operatorname{dom} \Delta \to \mathsf{AExp}_{\Delta'}$ by

$$g(x) = z \qquad\qquad g(u) = u,\;\; u \in \operatorname{dom} \Delta - \{x\}. \tag{1}$$

Define $h : \operatorname{dom} \Delta' \to \mathsf{AExp}_\Delta$ by

$$h(z) = x \qquad h(x) = v \qquad h(u) = u,\;\; u \in \operatorname{dom} \Delta' - \{z, x\}. \tag{2}$$

Extend h uniquely to a homomorphism $h : \mathsf{AExp}_{\Delta'} \to \mathsf{AExp}_\Delta$ by inductively defining $h(u.y) = h(u).y$ for $y \in \operatorname{dom} \Gamma$, where $\Delta' \vdash u : \mathcal{C}(\prod \Gamma)$. Likewise, extend g uniquely to a homomorphism $g : \mathsf{AExp}_\Delta \to \mathsf{AExp}_{\Delta'}$. Define a new set of axioms on $\mathsf{AExp}_{\Delta'}$:

$$\alpha' = \{(x, g(v))\} \cup \{(g(s), g(t)) \mid (s, t) \in \alpha\}. \tag{3}$$

Lemma 4.1 *Modulo α and α', the homomorphisms g and h are well defined and are inverses, thus the quotient algebras $\mathsf{AExp}_\Delta / \alpha$ and $\mathsf{AExp}_{\Delta'} / \alpha'$ are isomorphic.*

Proof. First we observe that h is a left inverse of g:

$$h(g(x)) = h(z) = x \qquad h(g(u)) = h(u) = u,\;\; u \in \operatorname{dom} \Delta - \{x\}.$$

Moreover, g is a left inverse of h modulo α':

$$g(h(z)) = g(x) = z \qquad g(h(u)) = g(u) = u,\;\; u \in \operatorname{dom} \Delta' - \{z, x\},$$

and since $(x, g(v))$ is an axiom of α' and $g(h(x)) = g(v)$,

$$\alpha' \vdash g(h(x)) = x.$$

Since h is a left inverse of g on generators $\operatorname{dom} \Delta$ of AExp_Δ, and since h and g are homomorphisms, h is a left inverse of g on all elements of AExp_Δ. Similarly, g is a left inverse of h modulo α' on all elements of $\mathsf{AExp}_{\Delta'}$.

Now we claim that

$$\alpha' \vdash s = t \;\;\Rightarrow\;\; \alpha \vdash h(s) = h(t), \tag{4}$$

thus h is well-defined modulo α and α'. By general considerations of universal algebra, it suffices to show that (4) holds for the axioms $(s,t) \in \alpha'$. For the axiom $(x, g(v))$, we wish to show $\alpha \vdash h(x) = h(g(v))$. This follows immediately from the facts that $h(x) = v$ and h is a left-inverse of g. For the axioms $(g(s), g(t))$ for $(s,t) \in \alpha$, we have $\alpha \vdash s = t$, and since h is a left-inverse of g, $\alpha \vdash h(g(s)) = h(g(t))$.

We have shown that h composed with the canonical map $\mathsf{AExp}_\Delta \to \mathsf{AExp}_\Delta/\alpha$ is well-defined on α'-congruence classes, therefore reduces to a homomorphism

$$h' : \mathsf{AExp}_{\Delta'}/\alpha' \to \mathsf{AExp}_\Delta/\alpha. \tag{5}$$

Likewise, one can show that $\alpha \vdash s = t$ implies $\alpha' \vdash g(s) = g(t)$ by the same argument, thus g reduces to a homomorphism

$$g' : \mathsf{AExp}_\Delta/\alpha \to \mathsf{AExp}_{\Delta'}/\alpha'. \tag{6}$$

Finally, since h is a left inverse of g and g is a left inverse of h modulo α', it follows that g' and h' are inverses, thus constitute an isomorphism between $\mathsf{AExp}_\Delta/\alpha$ and $\mathsf{AExp}_{\Delta'}/\alpha'$. □

Lemma 4.1 allows us to define the semantics of assignment to a creative variable:

$$\langle x := v,\, \Delta,\, \sigma,\, \alpha \rangle \;\to\; \langle (),\, \Delta[z/\Delta(x)],\, \sigma',\, \alpha' \rangle,$$

where z is fresh, $\sigma' = \sigma \circ h'$, and α' and h' are as defined in (3) and (5), respectively.

4.1.5 Assignment to Creative Fields

Now we treat the case of an assignment $u.y := v$, where both u and $u.y$ are creative. As before, we need to ensure that $u.y$ is not involved in the axiomatization α of the quotient structure so that the assignment will have no unintended consequences. However, unlike the previous case, if $\alpha \vdash u = v$, then assigning to $u.y$ also assigns the same value to $v.y$ due to the aliasing. Moreover, there is not necessarily an isomorphism between the two structures.

We first put α into the reduced form of Lemma 3.1. Let X be the set defined in that lemma. We can find variables $x, z, w \in X$ such that $\alpha \vdash u = x$, $\alpha \vdash v = w$, and $(x.y, z) \in \alpha$. We then define

$$\langle u.y := v,\, \Delta,\, \sigma,\, \alpha \rangle \;\to\; \langle (),\, \Delta,\, \sigma',\, \alpha' \rangle$$

where $\alpha' = (\alpha - \{(x.y, z)\}) \cup \{(x.y, w)\}$ and σ' is defined to agree with σ on all constructive expressions of the form r or $r.s$, where r is a variable. By the form of the reduced presentation, this determines σ' completely.

4.1.6 Other Small-Step Rules

(i) $\langle x, \Delta, \sigma, \alpha\rangle \to \langle \sigma([x]_\alpha), \Delta, \sigma, \alpha\rangle,\ \sigma([x]_\alpha) \neq \bot$, x constructive

(ii) $\langle x.y, \Delta, \sigma, \alpha\rangle \to \langle \sigma([x.y]_\alpha), \Delta, \sigma, \alpha\rangle,\ x.y$ constructive

(iii) $\langle f\ c, \Delta, \sigma, \alpha\rangle \to \langle f(c), \Delta, \sigma, \alpha\rangle$

(iv) $\langle u = v, \Delta, \sigma, \alpha\rangle \to \langle 1, \Delta, \sigma, \alpha\rangle, \alpha \vdash u = v$

(v) $\langle u = v, \Delta, \sigma, \alpha\rangle \to \langle 0, \Delta, \sigma, \alpha\rangle, \alpha \nvdash u = v$

(vi) $\langle \pi_y((v_x \mid x \in \mathrm{dom}\ \Gamma)), \Delta, \sigma, \alpha\rangle \to \langle v_y, \Delta, \sigma, \alpha\rangle$

(vii) $\langle [g_x \mid x \in \mathrm{dom}\ \Gamma]\,(\iota_y\, v), \Delta, \sigma, \alpha\rangle \to \langle (g_y\ v), \Delta, \sigma, \alpha\rangle$

Defined rules are

(viii) $\langle () \,;\, e, \Delta, \sigma, \alpha\rangle \to \langle e, \Delta, \sigma, \alpha\rangle$

(ix) $\langle \mathsf{if}\ 1\ \mathsf{then}\ d\ \mathsf{else}\ e, \Delta, \sigma, \alpha\rangle \to \langle d, \Delta, \sigma, \alpha\rangle$

(x) $\langle \mathsf{if}\ 0\ \mathsf{then}\ d\ \mathsf{else}\ e, \Delta, \sigma, \alpha\rangle \to \langle e, \Delta, \sigma, \alpha\rangle$

(xi) $\langle \mathsf{while}\ b\ \mathsf{do}\ e, \Delta, \sigma, \alpha\rangle \to \langle \mathsf{if}\ b\ \mathsf{then}\ (e\,;\,\mathsf{while}\ b\ \mathsf{do}\ e)\ \mathsf{else}\ (), \Delta, \sigma, \alpha\rangle$

The proviso "$\sigma([x]_\alpha) \neq \bot$" in (i) effectively makes x irreducible when this property holds. This is to allow Landin's knot to form self-referential terms. Recall that $\mathsf{let\,rec}\ x = d\ \mathsf{in}\ e$ abbreviates $\mathsf{let}\ x = \bot\ \mathsf{in}\ (x := d)\,;\,e$. The object $\bot$ is meant for this purpose only, and is not meant to be visible as the final value of a computation. In a real implementation one would prevent $\bot$ from becoming visible by imposing syntactic guardedness conditions on the form of d, as done for example in OCaml, or by raising a runtime error if the value of $\bot$ is ever required in the evaluation of d.

5 Applications

Nonces

A *nonce* is a creative object of type $\mathcal{C}(\mathbb{1})$. These are objects with no fields. They can be used as unique identifiers. We illustrate the use of nonces as variables in §6.

Records

A record with fields of type Γ is an object of type $\mathcal{C}(\prod\Gamma)$. Note that this is different from $\prod\Gamma$. The difference is that if $x_1 = y_1$ and $x_2 = y_2$, then $(x_1, x_2) = (y_1, y_2)$, whereas there can be distinct creative objects x and y with $x.1 = y.1$ and $x.2 = y.2$.

References

A reference is a record with a single field named !. The type of the reference is $\mathcal{C}(\prod \Gamma)$, where dom $\Gamma = \{!\}$, and $\Gamma(!)$ is the type of the datum. For example, an integer reference, which would be represented by the type int ref in OCaml, would have $\Gamma(!) = \mathbb{Z}$. The following OCaml expressions would translate to our language as indicated:

OCaml	*our language*
let $x =$ ref 3 in ...	let $x =$ new $\Gamma(3)$ in ...
$!x$	$x.!$
$x := 4$	$x.! := 4$

Arrays

An integer array of length m is a record with fields $\{0, 1, \ldots, m-1, \mathit{length}\}$. This would have type $\mathcal{C}(\prod \Gamma)$, where dom $\Gamma = \{0, 1, \ldots, m-1, \mathit{length}\}$, $\Gamma(i) = \mathbb{Z}$ for $0 \leq i \leq m-1$, and $\Gamma(\mathit{length}) = \mathbb{N}$. The following Java expressions would translate to our language as indicated:

Java	*our language*
int[] $x =$ new int[3];	let $x =$ new $\Gamma(0, \ldots, 0, 3)$ in ...
x.length	$x.\mathit{length}$
$x[0]$	$x.0$
$x[2] = x[3];$	$x.2 := x.3$

Objects

Objects (in the sense of object-oriented programming) present no difficulties. A creative type $\mathcal{C}(\prod \Gamma)$ can be regarded as a class with fields whose types are specified by Γ. If *self* is a variable of type $\Delta(\mathit{self}) = \mathcal{C}(\prod \Gamma)$, then other fields $x \in$ dom Γ of the object can be accessed from within the object as $\mathit{self}.x$. To create a new object of the class, we would say

$$\text{let rec } \mathit{self} = \text{new } \Gamma(v) \text{ in } \mathit{self} \tag{7}$$

The value of this expression is a new object in which the references to *self* in v have been backpatched via Landin's knot to refer to the object just created. If we like, we can even have $\mathit{self} \in$ dom Γ with $\Gamma(\mathit{self}) = \mathcal{C}(\prod \Gamma)$. The component of v corresponding to *self* should be *self*. In order to have $\Gamma(\mathit{self}) = \mathcal{C}(\prod \Gamma)$, the type must be coinductive.

Note that the use of Landin's knot is essential here. The traditional approach involving fixpoint combinators does not work, as fixpoint combinators do not interact correctly with new. Fixpoint combinators unwind a recursive definition syntactically, which would spawn a separate call to new with each recursive access.

Here is an example to demonstrate (7). Let dom $\Gamma = \{self, f, n\}$ with

$$\Gamma(self) = \mathcal{C}(\prod \Gamma) \qquad \Gamma(f) = \mathbb{N} \to () \qquad \Gamma(n) = \mathbb{N}.$$

Let us evaluate (7) with $v = (self,\ \lambda y.(self.n := y),\ 3)$. Substituting the definitions of let rec and let, we have

$$\begin{aligned}
&\mathsf{let\ rec}\ self = \mathsf{new}\ \Gamma(self,\ \lambda y.(self.n := y),\ 3)\ \mathsf{in}\ self \\
&\quad = \mathsf{let}\ self = \bot\ \mathsf{in}\ (self := \mathsf{new}\ \Gamma(self,\ \lambda y.(self.n := y),\ 3))\,;\ self \\
&\quad = (\lambda self.(self := \mathsf{new}\ \Gamma(self,\ \lambda y.(self.n := y),\ 3))\,;\ self)\ \bot.
\end{aligned}$$

Evaluating this expression in a state with Δ, σ, and α would result in the state

$$\langle (x := \mathsf{new}\ \Gamma(x,\ \lambda y.(x.n := y),\ 3))\,;\ x,\ \Delta',\ \sigma,\ \alpha' \rangle$$

where x is fresh, $\Delta' = \Delta[x/\mathcal{C}(\prod \Gamma)]$, and $\alpha' = \alpha \cup \{(x, \bot)\}$. One more step of the evaluation would yield

$$\langle (x := v)\,;\ x,\ \Delta'',\ \sigma',\ \alpha'' \rangle$$

where v is fresh and

$$\Delta'' = \Delta'[v/\mathcal{C}(\prod \Gamma)] \quad \sigma' = \sigma[v.f/\lambda y.(x.n := y)][v.n/3] \quad \alpha'' = \alpha' \cup \{(v.self, x)\}.$$

Now performing the assignment leaves the expression x and changes the aliasing relation to $(\alpha'' - \{(x, \bot)\}) \cup \{(x, v)\}$. Applying Lemma 3.1 with $x \in X$ and collecting garbage, we are left with the final state

$$\langle x,\ \Delta[x/\mathcal{C}(\prod \Gamma)],\ \sigma[x.f/\lambda y.(x.n := y)][x.n/3],\ \alpha \cup \{(x.self, x)\} \rangle.$$

To accommodate *nominal* classes in the sense of [19, §19.3], one could augment the new construct to allow new $C(e)$, where $C = \mathcal{C}(\prod \Gamma)$ is a class declaration, although we have not done so here.

6 Substitution and α-Conversion

In this section we demonstrate how syntactic equivalence of computational states gives rise to indiscernability in the semantic domain. We show how to model λ-terms semantically as elements of a coinductive datatype in which variables are nonces. In the semantic domain, α-conversion is an idempotent operation; that is, α-converting twice is the same as α-converting once. Equational reasoning using the aliasing relation α plays a large role in our arguments.

A λ-term is either a λ-variable, an application, or an abstraction. An application is a pair of λ-terms, an abstraction consists of a λ-variable (the parameter) and a λ-term (the body), and λ-variables are nonces. We can thus model

λ-terms with the coinductive type

$$\begin{array}{rl} \lambda\mathsf{Term} = \lambda\mathsf{Var} + \lambda\mathsf{App} + \lambda\mathsf{Abs} & \lambda\text{-coterms} \\ \lambda\mathsf{App} = \lambda\mathsf{Term} \times \lambda\mathsf{Term} & \text{applications} \\ \lambda\mathsf{Abs} = \lambda\mathsf{Var} \times \lambda\mathsf{Term} & \text{abstractions} \\ \lambda\mathsf{Var} = \mathcal{C}(\mathbb{1}) & \lambda\text{-variables} \end{array}$$

The type also contains λ-coterms (infinitary λ-terms), although they do not figure in our development.

The *free variables* of a λ-term are defined inductively by

$$\mathsf{FV}(y) = \{y\} \quad \mathsf{FV}(t_1\, t_2) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \quad \mathsf{FV}(\lambda y.t_0) = \mathsf{FV}(t_0) - \{y\}$$

They can be computed (for well-founded terms) by the following recursive program:

$$\begin{array}{l} \mathsf{let\ rec\ isFreeIn}\ (x : \lambda\mathsf{Var})\ (t : \lambda\mathsf{Term}) : \mathbb{2} = \\ \quad \mathsf{case}\ t\ \mathsf{of} \\ \quad\quad \mid \iota_0\, y \to y = x \\ \quad\quad \mid \iota_1\ (t_1, t_2) \to \mathsf{isFreeIn}\, x\, t_1 \vee \mathsf{isFreeIn}\, x\, t_2 \\ \quad\quad \mid \iota_2\ (y, t_0) \to y \neq x \wedge \mathsf{isFreeIn}\, x\, t_0 \end{array}$$

Likewise, *safe* (*capture-avoiding*) *substitution* is defined as a fixpoint of a system of equations. The result of substituting e for x in t is denoted $t[x/e]$ and is defined inductively by

$$y[x/e] = \begin{cases} e & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(t_1\, t_2)[x/e] = (t_1[x/e]\ \ t_2[x/e])$$

$$(\lambda y.t_0)[x/e] = \begin{cases} \lambda y.t_0 & \text{if } y = x \\ \lambda y.(t_0[x/e]) & \text{if } y \neq x \text{ and } y \notin \mathsf{FV}(e) \\ \lambda z.(t_0[y/z][x/e]) & \text{otherwise, where } z \notin \{x\} \cup \mathsf{FV}(t_0) \cup \mathsf{FV}(e) \end{cases}$$

In the last rule, to satisfy the proviso $z \notin \{x\} \cup \mathsf{FV}(t_0) \cup \mathsf{FV}(e)$, it suffices to take z fresh. This leads to the following recursive program:

$$\begin{array}{l} \mathsf{let\ rec\ subst}\ (t : \lambda\mathsf{Term})\ (x : \lambda\mathsf{Var})\ (e : \lambda\mathsf{Term}) : \lambda\mathsf{Term} = \\ \quad \mathsf{case}\ t\ \mathsf{of} \\ \quad\quad \mid \iota_0\, y \to \mathsf{if}\ y = x\ \mathsf{then}\ e\ \mathsf{else}\ t \\ \quad\quad \mid \iota_1\ (t_1, t_2) \to \iota_1\ (\mathsf{subst}\, t_1\, x\, e, \mathsf{subst}\, t_2\, x\, e) \\ \quad\quad \mid \iota_2\ (y, t_0) \to \mathsf{if}\ y = x\ \mathsf{then}\ t \\ \quad\quad\quad\quad\quad\quad \mathsf{else\ if}\ \neg(\mathsf{isFreeIn}\, y\, e)\ \mathsf{then}\ \iota_2\ (y, \mathsf{subst}\, t_0\, x\, e) \\ \quad\quad\quad\quad\quad\quad \mathsf{else\ let}\ z = \mathsf{new}\ \lambda\mathsf{Var}\ \mathsf{in}\ \iota_2\ (z, \mathsf{subst}\ (\mathsf{subst}\, t_0\, y\, (\iota_0\, z))\, x\, e) \end{array}$$

If e is a variable w, this simplifies to

$$y[x/w] = \begin{cases} w & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(t_1\, t_2)[x/w] = (t_1[x/w])\,(t_2[x/w])$$

$$(\lambda y.t_0)[x/w] = \begin{cases} \lambda y.t_0 & \text{if } y = x \\ \lambda y.(t_0[x/w]) & \text{if } y \neq x \text{ and } y \neq w \\ \lambda z.(t_0[y/z][x/w]) & \text{if } w = y \neq x, \text{where } z \notin \{x, w\} \cup \mathsf{FV}(t_0). \end{cases}$$

$$\begin{array}{l} \mathsf{let\ rec\ subst'}\ (t : \lambda\mathsf{Term})\ (x : \lambda\mathsf{Var})\ (w : \lambda\mathsf{Var}) : \lambda\mathsf{Term} = \\ \quad \mathsf{case}\ t\ \mathsf{of} \\ \quad\quad \mid \iota_0\, y \to \mathsf{if}\ y = x\ \mathsf{then}\ \iota_0\, w\ \mathsf{else}\ t \\ \quad\quad \mid \iota_1\ (t_1, t_2) \to \iota_1\ (\mathsf{subst'}\ t_1\, x\, w,\ \mathsf{subst'}\ t_2\, x\, w) \\ \quad\quad \mid \iota_2\ (y, t_0) \to \mathsf{if}\ y = x\ \mathsf{then}\ t \\ \quad\quad\quad\quad\quad\quad \mathsf{else\ if}\ y \neq w\ \mathsf{then}\ \iota_2\ (y,\ \mathsf{subst'}\ t_0\, x\, w) \\ \quad\quad\quad\quad\quad\quad \mathsf{else\ let}\ z = \mathsf{new}\ \lambda\mathsf{Var}\ \mathsf{in}\ \iota_2\ (z,\ \mathsf{subst'}\ (\mathsf{subst'}\ t_0\, y\, z)\, x\, w) \end{array}$$

Lemma 6.1 *Modulo α-equivalence and garbage collection, the following big-step rules are sound:*

$$\frac{\alpha \vdash x = y}{\langle \mathsf{subst'}\ (\iota_0\, y)\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \ \to\ \langle \iota_0\, v,\ \Delta,\ \sigma,\ \alpha\rangle} \tag{8}$$

$$\frac{\alpha \nvdash x = y}{\langle \mathsf{subst'}\ (\iota_0\, y)\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \ \to\ \langle \iota_0\, y,\ \Delta,\ \sigma,\ \alpha\rangle} \tag{9}$$

$$\frac{\langle \mathsf{subst'}\ e_0\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle v_0,\ \Delta,\ \sigma,\ \alpha\rangle \quad \langle \mathsf{subst'}\ e_1\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle v_1,\ \Delta,\ \sigma,\ \alpha\rangle}{\langle \mathsf{subst'}\ (\iota_1\ (e_0, e_1))\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_1\ (v_0, v_1),\ \Delta,\ \sigma,\ \alpha\rangle} \tag{10}$$

$$\frac{\alpha \vdash x = y}{\langle \mathsf{subst'}\ (\iota_2\ (y, t))\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_2\ (y, t),\ \Delta,\ \sigma,\ \alpha\rangle} \tag{11}$$

$$\frac{\alpha \nvdash x = y \quad \alpha \nvdash y = v \quad \langle \mathsf{subst'}\ t\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle u,\ \Delta,\ \sigma,\ \alpha\rangle}{\langle \mathsf{subst'}\ (\iota_2\ (y, t))\, x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_2\ (y, u),\ \Delta,\ \sigma,\ \alpha\rangle} \tag{12}$$

Proof. We start with rule (8). Suppose $\alpha \vdash y = x$. Let

$$\begin{gathered} \Delta' = \Delta[t'/\lambda\mathsf{Term}][x'/\lambda\mathsf{Var}][v'/\lambda\mathsf{Var}] \qquad \Delta'' = \Delta'[y'/\lambda\mathsf{Var}] \\ \alpha' = \alpha \cup \{(x, x'),\ (v, v')\} \qquad \alpha'' = \alpha' \cup \{(y, y')\} \\ \sigma' = \sigma[t'/\iota_0\, y], \end{gathered} \tag{13}$$

where t', x', v', y' are fresh. We will first give the steps of the derivation, then

give a brief justification of each step afterwards.

$$
\begin{aligned}
&\langle \mathsf{subst}'\ (\iota_0\, y)\ x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \\
&\quad\to \langle (\lambda t x w.[\lambda y.\mathsf{if}\ y = x\ \mathsf{then}\ \iota_0\, w\ \mathsf{else}\ \iota_0\, y, \ldots]\ t)\ (\iota_0\, y)\ x\, v,\ \Delta,\ \sigma,\ \alpha\rangle && (14)\\
&\quad\to \langle [\lambda y.\mathsf{if}\ y = x'\ \mathsf{then}\ \iota_0\, v'\ \mathsf{else}\ \iota_0\, y, \ldots]\ t',\ \Delta',\ \sigma',\ \alpha'\rangle && (15)\\
&\quad\to \langle \mathsf{if}\ y' = x'\ \mathsf{then}\ \iota_0\, v'\ \mathsf{else}\ \iota_0\, y',\ \Delta'',\ \sigma',\ \alpha''\rangle && (16)\\
&\quad\to \langle \iota_0\, v',\ \Delta'',\ \sigma',\ \alpha''\rangle && (17)\\
&\quad= \langle \iota_0\, v,\ \Delta'',\ \sigma',\ \alpha''\rangle && (18)\\
&\quad= \langle \iota_0\, v,\ \Delta,\ \sigma,\ \alpha\rangle. && (19)
\end{aligned}
$$

For (14), we have just replaced subst' with its definition. This is just an application of small-step rule (i) of §4.1.6.

We obtain (15) from (14) by doing three successive function applications as defined in §4.1.1. The first allocates a fresh constructive variable t' of type $\lambda\mathsf{Term}$, substitutes it for t in the body of the function, and binds it to the argument $\iota_0\, y$ in σ to get σ'. The second and last allocate fresh creative variables x' and v' of type $\lambda\mathsf{Var}$, substitute them for x and w, respectively, in the body of the function, and equate them to the arguments x and v, respectively, thereby extending α to α'. The new type environment is Δ'.

We obtain (16) from (15) by rule (vii) of §4.1.6, the small-step rule for the case statement. After lookup of t', its value $\iota_0\, y$ is analyzed and the function corresponding to index 0 in the tuple (the one shown) is dispatched. That function is applied to y, which causes a fresh creative variable y' of type $\lambda\mathsf{Var}$ to be allocated, substituted for y in the body, and equated with the argument y in α' to get α''. The new type environment is Δ''.

For (17), since $\alpha \vdash y = x$ by assumption, we have $\alpha'' \vdash y' = x'$, therefore the conditional test succeeds, resulting in the value $\iota_0\, v'$. Since $\alpha'' \vdash v = v'$, (17) is equivalent to (18). Finally, (19) is obtained by garbage collection, observing that t', x', v', and y' are no longer accessible from $\iota_0\, v$.

The proof of rule (9) is very similar, except that at step (17) we obtain $\iota_0\, y'$ instead of $\iota_0\, v'$ because $\alpha'' \not\vdash y' = x'$.The proof of rule (11) is also very similar.

For rule (10), let

$$
\begin{gathered}
\Delta' = \Delta[t'/\lambda\mathsf{Term}][x'/\lambda\mathsf{Var}][v'/\lambda\mathsf{Var}] \qquad \Delta'' = \Delta'[y'/\lambda\mathsf{App}] \\
\alpha' = \alpha \cup \{(x, x'),\ (v, v')\} \\
\sigma' = \sigma[t'/\iota_1\ (e_0, e_1)] \qquad \sigma'' = \sigma'[y'/(e_0, e_1)],
\end{gathered}
$$

where t', x', v', y' are fresh. By reasoning similar to the above, we have

$$
\begin{aligned}
&\langle \mathsf{subst}'\ (\iota_1\ (e_0, e_1))\ x\, v,\ \Delta,\ \sigma,\ \alpha\rangle \\
&\quad\to \langle [\ldots, \lambda y.\iota_1\ (\mathsf{subst}'\ (\pi_0\, y)\ x'\, v',\ \mathsf{subst}'\ (\pi_1\, y)\ x'\, v'), \ldots]\ t',\ \Delta',\ \sigma',\ \alpha'\rangle \\
&\quad\to \langle \iota_1\ (\mathsf{subst}'\ (\pi_0\, y')\ x'\, v',\ \mathsf{subst}'\ (\pi_1\, y')\ x'\, v'),\ \Delta'',\ \sigma'',\ \alpha'\rangle \\
&\quad= \langle \iota_1\ (\mathsf{subst}'\ (\pi_0\, y')\ x\, v,\ \mathsf{subst}'\ (\pi_1\, y')\ x\, v),\ \Delta'',\ \sigma'',\ \alpha'\rangle.
\end{aligned}
$$

The last equation follows from the fact that $\alpha' \vdash x = x'$ and $\alpha' \vdash v = v'$. Now evaluating $\pi_0\, y'$ gives e_0, and by the left-hand premise of (10), $\mathsf{subst}'\, e_0\, x\, v$ reduces to v_0 in context. Similarly, by the right-hand premise, $\mathsf{subst}'\, (\pi_0\, y')\, x\, v$ reduces to v_1 in context. This leaves us with

$$\langle \iota_1\, (v_0, v_1),\, \Delta'',\, \sigma',\, \alpha'' \rangle = \langle \iota_1\, (v_0, v_1),\, \Delta,\, \sigma,\, \alpha \rangle,$$

where the right-hand side is obtained from the left by garbage collection.

Finally, for rule (12), let

$$\Delta' = \Delta[t'/\lambda\mathsf{Term}][x'/\lambda\mathsf{Var}][v'/\lambda\mathsf{Var}][y'/\lambda\mathsf{Abs}]$$
$$\alpha' = \alpha \cup \{(x, x'),\, (v, v')\} \qquad \sigma' = \sigma[t'/\iota_2\, (y, t)][y'/(y, t)],$$

where t', x', v', y' are fresh. As above, we have

$$\begin{aligned}
&\langle \mathsf{subst}'\, (\iota_2\, (y, t))\, x\, v,\, \Delta,\, \sigma,\, \alpha \rangle \\
&\quad \to \langle \iota_2\, (\pi_0\, y',\, \mathsf{subst}'\, (\pi_1\, y')\, x'\, v'),\, \Delta',\, \sigma',\, \alpha' \rangle \\
&\quad \to \langle \iota_2\, (y,\, \mathsf{subst}'\, t\, x'\, v'),\, \Delta',\, \sigma',\, \alpha' \rangle \\
&\quad = \langle \iota_2\, (y,\, \mathsf{subst}'\, t\, x\, v),\, \Delta,\, \sigma,\, \alpha \rangle && (20) \\
&\quad \to \langle \iota_2\, (y, u),\, \Delta,\, \sigma,\, \alpha \rangle. && (21)
\end{aligned}$$

with (20) from the fact that $\alpha' \vdash x = x'$, $\alpha' \vdash v = v'$, and garbage collection, and (21) from the premise of (12) applied in context. □

Lemma 6.2 *Let $\Delta \vdash e : \lambda\mathsf{Term}$ and $x, u, v \in \mathrm{dom}\, \Delta$. Assume that $\alpha \not\vdash y = u$ and $\alpha \not\vdash y = v$ for $y = x$ or any y occurring in e. The states*

$$\langle \mathsf{subst}'\, (\mathsf{subst}'\, e\, x\, u)\, u\, v,\, \Delta,\, \sigma,\, \alpha \rangle \qquad\qquad \langle \mathsf{subst}'\, e\, x\, v,\, \Delta,\, \sigma,\, \alpha \rangle$$

reduce to equivalent states modulo α-equivalence and garbage collection.

Proof. For the case $e = \iota_0\, y$ and $\alpha \vdash y = x$, by rule (8) both states reduce to $\langle \iota_0\, v,\, \Delta,\, \sigma,\, \alpha \rangle$:

$$\begin{aligned}
\langle \mathsf{subst}'\, (\mathsf{subst}'\, (\iota_0\, y)\, x\, u)\, u\, v,\, \Delta,\, \sigma,\, \alpha \rangle &\to \langle \mathsf{subst}'\, (\iota_0\, u)\, u\, v,\, \Delta,\, \sigma,\, \alpha \rangle \\
&\to \langle \iota_0\, v,\, \Delta,\, \sigma,\, \alpha \rangle \\
\langle \mathsf{subst}'\, (\iota_0\, y)\, x\, v,\, \Delta,\, \sigma,\, \alpha \rangle &\to \langle \iota_0\, v,\, \Delta,\, \sigma,\, \alpha \rangle.
\end{aligned}$$

If $\alpha \not\vdash y = x$, by rule (9) both states reduce to $\langle \iota_0\, y,\, \Delta,\, \sigma,\, \alpha \rangle$:

$$\begin{aligned}
\langle \mathsf{subst}'\, (\mathsf{subst}'\, (\iota_0\, y)\, x\, u)\, u\, v,\, \Delta,\, \sigma,\, \alpha \rangle &\to \langle \mathsf{subst}'\, (\iota_0\, y)\, u\, v,\, \Delta,\, \sigma,\, \alpha \rangle \\
&\to \langle \iota_0\, y,\, \Delta,\, \sigma,\, \alpha \rangle \\
\langle \mathsf{subst}'\, (\iota_0\, y)\, x\, v,\, \Delta,\, \sigma,\, \alpha \rangle &\to \langle \iota_0\, y,\, \Delta,\, \sigma,\, \alpha \rangle.
\end{aligned}$$

For the case $\iota_1\ (e_0, e_1)$, we have

$$\langle \mathsf{subst}'\ e_0\ x\ u,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_0',\ \Delta,\ \sigma,\ \alpha\rangle \quad \langle \mathsf{subst}'\ e_0'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_0'',\ \Delta,\ \sigma,\ \alpha\rangle$$
$$\langle \mathsf{subst}'\ e_1\ x\ u,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_1',\ \Delta,\ \sigma,\ \alpha\rangle \quad \langle \mathsf{subst}'\ e_1'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_1'',\ \Delta,\ \sigma,\ \alpha\rangle,$$

thus

$$\langle \mathsf{subst}'\ (\mathsf{subst}'\ e_0\ x\ u)\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \mathsf{subst}'\ e_0'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_0'',\ \Delta,\ \sigma,\ \alpha\rangle$$
$$\langle \mathsf{subst}'\ (\mathsf{subst}'\ e_1\ x\ u)\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \mathsf{subst}'\ e_1'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_1'',\ \Delta,\ \sigma,\ \alpha\rangle.$$

By the induction hypothesis,

$$\langle \mathsf{subst}'\ e_0\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_0'',\ \Delta,\ \sigma,\ \alpha\rangle \quad \langle \mathsf{subst}'\ e_1\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle e_1'',\ \Delta,\ \sigma,\ \alpha\rangle.$$

By rule (10),

$$\langle \mathsf{subst}'\ (\iota_1\ (e_0, e_1))\ x\ u,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_1\ (e_0', e_1'),\ \Delta,\ \sigma,\ \alpha\rangle$$
$$\langle \mathsf{subst}'\ (\iota_1\ (e_0', e_1'))\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_1\ (e_0'', e_1''),\ \Delta,\ \sigma,\ \alpha\rangle,$$

therefore

$$\begin{aligned}\langle \mathsf{subst}'\ (\mathsf{subst}'\ (\iota_1\ (e_0, e_1))\ x\ u)\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle &\to \langle \mathsf{subst}'\ (\iota_1\ (e_0', e_1'))\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \\ &\to \langle \iota_1\ (e_0'', e_1''),\ \Delta,\ \sigma,\ \alpha\rangle, \\ \langle \mathsf{subst}'\ (\iota_1\ (e_0', e_1'))\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle &\to \langle \iota_1\ (e_0'', e_1''),\ \Delta,\ \sigma,\ \alpha\rangle.\end{aligned}$$

For the case $\iota_2\ (y, t)$, if $\alpha \vdash y = x$, by rule (11) and the fact that $\alpha \nvdash u = w$ for any w occurring in t, we have

$$\begin{aligned}\langle \mathsf{subst}'\ (\mathsf{subst}'\ (\iota_2\ (y, t))\ x\ u)\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle &\to \langle \mathsf{subst}'\ (\iota_2\ (y, t))\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \\ &\to \langle \iota_2\ (y, t),\ \Delta,\ \sigma,\ \alpha\rangle \\ \langle \mathsf{subst}'\ (\iota_2\ (y, t))\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle &\to \langle \iota_2\ (y, t),\ \Delta,\ \sigma,\ \alpha\rangle.\end{aligned}$$

If $\alpha \nvdash y = x$, we have $\alpha \nvdash y = u$ and $\alpha \nvdash y = v$ by the assumptions of the lemma, and

$$\langle \mathsf{subst}'\ t\ x\ u,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle t',\ \Delta,\ \sigma,\ \alpha\rangle \quad \langle \mathsf{subst}'\ t'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle t'',\ \Delta,\ \sigma,\ \alpha\rangle,$$

thus

$$\langle \mathsf{subst}'\ (\mathsf{subst}'\ t\ x\ u)\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \mathsf{subst}'\ t'\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle t'',\ \Delta,\ \sigma,\ \alpha\rangle.$$

By the induction hypothesis,

$$\langle \mathsf{subst}'\ t\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle t'',\ \Delta,\ \sigma,\ \alpha\rangle.$$

By rule (12),

$$\langle \mathsf{subst}'\ (\iota_2\ (y, t))\ x\ u,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_2\ (y, t'),\ \Delta,\ \sigma,\ \alpha\rangle$$
$$\langle \mathsf{subst}'\ (\iota_2\ (y,\ t'))\ u\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_2\ (y, t''),\ \Delta,\ \sigma,\ \alpha\rangle$$
$$\langle \mathsf{subst}'\ (\iota_2\ (y, t))\ x\ v,\ \Delta,\ \sigma,\ \alpha\rangle \to \langle \iota_2\ (y, t''),\ \Delta,\ \sigma,\ \alpha\rangle,$$

therefore

$$\begin{aligned}\langle \mathsf{subst}'\,(\mathsf{subst}'\,(\iota_2\,(y,t))\,x\,u)\,u\,v,\,\Delta,\,\sigma,\,\alpha\rangle &\to \langle \mathsf{subst}'\,(\iota_2\,(y,t'))\,u\,v,\,\Delta,\,\sigma,\,\alpha\rangle\\ &\to \langle \iota_2\,(y,t''),\,\Delta,\,\sigma,\,\alpha\rangle.\end{aligned}$$

□

To α-convert, we would map $\lambda x.e$ to $\lambda z.(e[x/z])$, where $z \notin \mathsf{FV}(e) - \{x\}$. We choose $z \notin \mathsf{FV}(e) - \{x\}$ to avoid the capture of a free occurrences of z in e as a result of the renaming. Usually we would simply choose a fresh z.

In our language, this would be implemented by a function

$$\begin{aligned}&\mathsf{alpha} : \lambda\mathsf{Abs} \to \lambda\mathsf{Abs}\\ &\mathsf{alpha} = \lambda t.\mathsf{let}\ z = \mathsf{new}\ \lambda\mathsf{Var}\ \mathsf{in}\ (z,\,\mathsf{subst}'\,(\pi_1\,t)\,(\pi_0\,t)\,z),\end{aligned}$$

or more informally,

$$\mathsf{alpha}\ (x,e) = \mathsf{let}\ z = \mathsf{new}\ \lambda\mathsf{Var}\ \mathsf{in}\ (z,\,\mathsf{subst}'\,e\,x\,z).$$

The following theorem illustrates how syntactic equivalence of computational states gives rise to indiscernability in the semantic domain. It states that α-conversion is an idempotent operation; that is, performing it twice gives the same result as performing it once.

Theorem 6.3 *Modulo α-equivalence and garbage collection,*

$$\mathsf{alpha}\,(\mathsf{alpha}\ (x,e)) = \mathsf{alpha}\ (x,e).$$

Proof. In the evaluation of $\langle \mathsf{alpha}\ (x,e),\,\Delta,\,\sigma,\,\alpha\rangle$, let t,u,v be fresh variables and let

$$\Delta' = \Delta[t/\lambda\mathsf{Abs}] \qquad \sigma' = \sigma[t/(x,e)] \qquad \alpha' = \alpha \cup \{(u,v)\}.$$

Suppose

$$\langle \mathsf{subst}'\,e\,x\,u,\,\Delta[u/\lambda\mathsf{Var}],\,\sigma,\,\alpha\rangle \to \langle e',\,\Delta[u/\lambda\mathsf{Var}],\,\sigma,\,\alpha\rangle.$$

The evaluation yields the following sequence of states:

$$\begin{aligned}&\langle \mathsf{alpha}\ (x,e),\,\Delta,\,\sigma,\,\alpha\rangle\\ &\quad\to \langle \mathsf{let}\ z = \mathsf{new}\ \lambda\mathsf{Var}\ \mathsf{in}\ (z,\,\mathsf{subst}'\,(\pi_1\,t)\,(\pi_0\,t)\,z),\,\Delta',\,\sigma',\,\alpha\rangle\\ &\quad\to \langle (\lambda z.(z,\,\mathsf{subst}'\,(\pi_1\,t)\,(\pi_0\,t)\,z))\,v,\,\Delta'[v/\lambda\mathsf{Var}],\,\sigma',\rangle\\ &\quad\to \langle (u,\,\mathsf{subst}'\,(\pi_1\,t)\,(\pi_0\,t)\,u),\,\Delta'[v/\lambda\mathsf{Var}][u/\lambda\mathsf{Var}],\,\sigma',\,\alpha'\rangle\\ &\quad\to \langle (u,\,\mathsf{subst}'\,e\,x\,u),\,\Delta'[v/\lambda\mathsf{Var}][u/\lambda\mathsf{Var}],\,\sigma',\,\alpha'\rangle\\ &\quad= \langle (u,\,\mathsf{subst}'\,e\,x\,u),\,\Delta[u/\lambda\mathsf{Var}],\,\sigma,\,\alpha\rangle \qquad (22)\\ &\quad\to \langle (u,e'),\,\Delta[u/\lambda\mathsf{Var}],\,\sigma,\,\alpha\rangle.\end{aligned}$$

Step (22) is by garbage collection. Using this,

$$
\begin{aligned}
&\langle \mathsf{alpha}\,(\mathsf{alpha}\,(x,e)),\ \Delta,\ \sigma,\ \alpha\rangle \\
&\quad = \langle \mathsf{alpha}\,(u,e'),\ \Delta[u/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle \\
&\quad \to \langle (v,\ \mathsf{subst}'\ e'\ u\ v),\ \Delta[u/\lambda\mathsf{Var}][v/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle && (23)\\
&\quad = \langle (v,\ \mathsf{subst}'\ e\ x\ v),\ \Delta[u/\lambda\mathsf{Var}][v/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle && (24)\\
&\quad = \langle (v,\ \mathsf{subst}'\ e\ x\ v),\ \Delta[v/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle && (25)\\
&\quad = \langle (u,\ \mathsf{subst}'\ e\ x\ u),\ \Delta[u/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle && (26)\\
&\quad \to \langle (u,e'),\ \Delta[u/\lambda\mathsf{Var}],\ \sigma,\ \alpha\rangle.
\end{aligned}
$$

Step (23) is by the same argument as (22). Step (24) is by Lemma 6.2. Steps (25) and (26) are by garbage collection and renaming of a creative variable. □

7 Conclusion

We have shown how to model the creation of new indiscernible semantic objects during program execution and how to incorporate this device in a higher-order functional language with imperative and object-oriented features. Modeling indescernables is desirable because it abstracts away from properties that are only needed to allocate them from a preexisting set.

The explicit aliasing relation α facilitates equational reasoning about the state of a higher-order computation. However, much more needs to be done to develop and formalize this equational system.

Acknowledgments

Thanks to Robert Constable, Nate Foster, Jean-Baptiste Jeannin, Konstantinos Mamouras, Andrew Myers, Dirk Pattinson, Mark Reitblatt, Aaron Stump, and the members of the PLDG seminar at Cornell for stimulating discussions.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 2002.

[2] Stuart F. Allen. An abstract semantics for atoms in nuprl. Technical Report TR2006-2032, Cornell University, 2006.

[3] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing equivalence of regular languages. *J. Automata, Languages and Combinatorics*, 15(1–2):7–25, 2010.

[4] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[5] Mark Bickford and Robert Constable. Formal foundations of computer security. In Orna Grumberg, editor, *Formal Logical Methods for System Security and Correctness*, pages 29–52. IOS Press, 2008.

[6] Max Black. The identity of indiscernibles. *Mind*, 61(242):153–164, 1952.

[7] George Boolos. To be is to be a value of a variable (or to be some values of some variables). *J. Philosophy*, 81:430–450, 1984.

[8] Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: equational logic with names and binding. *J. Logic and Computation*, 19(6):1455–1508, December 2009.

[9] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3-5), July 2002.

[10] Ian Hacking. The identity of indiscernibles. *J. Philosophy*, 72(9):249–256, May 1975.

[11] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[12] Jean-Baptiste Jeannin. Capsules and closures. In Michael Mislove and Joël Ouaknine, editors, *Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII)*, pages 191–213, Pittsburgh, PA, May 2011. Elsevier Electronic Notes in Theoretical Computer Science.

[13] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. Technical Report `http://hdl.handle.net/1813/22082`, Computing and Information Science, Cornell University, January 2011.

[14] Jean-Baptiste Jeannin and Dexter Kozen. Capsules and separation. Technical Report `http://hdl.handle.net/1813/28284`, Computing and Information Science, Cornell University, January 2012.

[15] Dexter Kozen. Complexity of finitely presented algebras. In *Proc. 9th Symp. Theory of Comput.*, pages 164–177. ACM, May 1977.

[16] Dexter Kozen. Realization of coinductive types. In Michael Mislove and Joël Ouaknine, editors, *Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII)*, pages 148–155, Pittsburgh, PA, May 2011. Elsevier Electronic Notes in Theoretical Computer Science.

[17] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[18] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205:557–580, 2007.

[19] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] Andrew M. Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science: Proceedings of the 18th International Symposium MFCS '93*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.

[21] Andrew M. Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.