FEFERMAN-LANDIN LOGIC

IAN A. MASON AND CAROLYN L. TALCOTT

Abstract. This paper presents a logic based on Feferman's Variable Type Theories for reasoning about programs written in a class of imperative functional languages called Landinesque languages.

To Solomon Feferman on the occasion of his 70th Birthday with much gratitude.

§**1. Historical Background.** Feferman-Landin Logic is so named because of the influence that the work of both Feferman and Landin has had on our work. Of particular relevance are two key ideas:

> *Landin's idea*: a programming language can be thought of as the lambda calculus augmented by operations on the basic data, as well as computational environment. We call such languages *Landinesque languages*.
> *Feferman's idea*: to formalize constructive mathematics in 2-sorted classical theories called variable type theories. In these theories both functions and data are objects of discourse in a first order setting, as are collections of such things.

Both authors were PhD students of Solomon Feferman in the early 1980's where, under his influence, we began the research program leading to the work that this paper will report on. Both the theories used by Feferman, and his approach to formalizing constructive mathematics seemed very natural and we felt that this approach should also work for developing formal systems for reasoning about programs.

The first step in such an endeavor is to develop the mathematical semantics of the programs of interest. Our focus has been on reasoning about non-functional primitives. Together we have studied lambda calculus augmented by operations for manipulating control [39], manipulating memory [29], and for communicating asynchronously in an open distributed environment [2]. In each case we developed an operational semantics based solely on syntactic entities, and studied the natural operational equivalence generated by that semantics (operational equivalence meaning being equi-defined in all closing contexts). Crucial in such work is the ability to simplify the criteria for operational equivalence to some form of context lemma a la Robin Milner [31] to avoid the complexity of reasoning about arbitrary program

The first author was partially supported by UNE URG grant 6590 881 499 and UNE small ARC grant 8282 881 499.

The second author was partially supported by NSF CCR-9633419, DARPA/Rome Labs AF F30602-96-1-0300, DARPA/NASA NAS2-98073, and ONR N00012-99-C-0198.

The authors wish to thank Jonathan Ford and the anonymous referees for their helpful comments.

Meeting

contexts. In both the case of memory, and control, we have established such simplifications now known in the literature as *CIU* theorems (*C*losed *I*nstantiations of *U*ses). The case for communicating asynchronously in an open distributed environment remains open.

After developing this syntactic style of operational semantics, and the corresponding characterization of operational equivalence, we used this as a basis for developing logics based on Solomon Feferman's Variable typed approach. These systems are two sorted theories of operations and classes initially developed for the formalization of constructive mathematics [5, 6, 7, 8] and later applied to the study of purely functional languages [10, 11, 9].

Our work uses the syntactic operational semantics to develop expressive logics and goes well beyond traditional programming logics, such as Hoare's logic [3] and Dynamic logic [21] by treating a richer language and expressing more properties. It is close in spirit to Specification Logic [37] and to Evaluation Logic [34]. Particular logics, with corresponding reasoning principles, have been presented for the case of manipulating control [41], and for the case of manipulating memory [22], for which it is possible to establish a limited form of completeness [26].

The work discussed above studied specific Landinesque languages in a somewhat ad hoc fashion. More recently the second author has unified the ideas emerging from our ad hoc approach to syntactic operational semantics by developing a general theory of Landinesque languages [42]. The approach starts with a small step semantics in which computation state is represented using syntactic entities such as expressions and contexts. There is a single reduction rule for each operation. Care is taken so that the reduction rule for an operation is not changed when new operations or new pieces of state are added. Computation is uniform is the sense that reduction steps can be performed on states with missing parts and the missing information can be filled in later. Such a syntactic reduction system has the combined advantages of a simple transition system semantics and the symbolic reasoning of a reduction calculus. The main result is the CIU theorem for languages with uniform semantics, that simplifies reasoning about operational equivalence restricting the contexts that must be considered.

In this paper we build upon this uniform approach to present a variable typed logic for such languages, thus extending our previous work on specifying and reasoning about programs to a wide class of imperative functional languages. As such this paper subsumes the case studies presented concerning operations that manipulate control, as well as those concerning operations that manipulate memory.

The plan of this paper is as follows. In the next section we present a summary and simplification of the core results of [42]. The form and presentation of the results presented in this section has benefitted greatly from the formalization, in PVS, of this approach by Jonathan Ford at the University of New England. This formalization is reported in [15]. We then present the syntax and semantics of our variable typed logic for this general framework. We do this in two stages. We first present the first-order semantics in the third section, and then in the fourth section concentrate on the notion of classes. Section five deals with the general principles that hold for these uniform languages, while section six presents a detailed example of a particular language that incorporates both control and memory manipulation. The final section describes our conclusions and further directions that this research may take.

Notation. We conclude the introduction with a summary of our notation conventions. Let X, X_0, X_1 be sets. We specify meta-variable conventions in the form: let x range over X, which should be read as: the meta-variable x and decorated variants such as x', x_0, $\ldots$, range over the set X. $\mathrm{Fmap}[X_0, X_1]$ is the set of finite maps from X_0 to X_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f, $f\{x \mapsto x'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{x\}$, $f'(x) = x'$, and $f'(z) = f(z)$ for $z \neq x, z \in \mathrm{Dom}(f)$. Also $f\lceil X$ is the restriction of f to X: the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cap X$ and $f'(x) = f(x)$ for $x \in \mathrm{Dom}(f')$. $\mathbf{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers and $i, j, n, n_0, \ \ldots$ range over $\mathbf{N}$. In the defining equations for various syntactic classes we use two notational conventions: pointwise lifting of syntax operations to syntax classes; and the Einstein summation convention that a phrase of the form $F_n(Z^n)$ abbreviates $\bigcup_{n\in\mathbf{N}} F_n(Z^n)$. For example if Ω is a ranked set of operator symbols, then the terms over Ω can be defined inductively by (as the least solution to) the equation: $T_\Omega = \Omega_n(T_\Omega^n)$. Unabbreviated, this equation reads:

$$T_\Omega = \bigcup_{n\in\mathbf{N}} \{\omega(t_1, \ldots , t_n) \mid \omega \in \Omega_n \wedge t_i \in T_\Omega \quad \text{for} \quad 1 \leq i \leq n\}.$$

§**2. Uniform Landinesque Languages.** In this section a general framework for studying the semantics of λ-languages is set up along the lines of [42], where detailed proofs of some of the claims can be found. The syntactic entities and semantic notions of λ-languages are defined and the properties required for a uniform semantics are stated. Then several results, including the CIU theorem, valid in any λ-language with uniform semantics are presented. This is the mathematical theory that we wish to formalize following Feferman's approach. We begin with an overview of the concepts and results.

A small-step operational semantics is obtained by defining a notion of state and a single step reduction relation on states. States consist of an expression and a state context. A state context often describes dynamically created entities such as memory cells, arrays, files, etc. The form of state contexts needed depends on the choice of primitive operations. There is an empty state context, and for each state there is an associated expression representing that state. Value expressions are a subset of the set of expressions used to represent semantic values. These include variables, atoms, and lambdas. If the expression component of a state is a value, then the state is a value state and no reduction steps are possible. Otherwise, the expression decomposes uniquely into a redex placed in a reduction context. A (call-by-value) redex is a primitive operator applied to a list of values. There is one reduction rule for each primitive operator, and the single-step reduction relation on states is determined by the reduction rule for the redex operator. Of course it may happen that a redex is ill-formed (a runtime error) and no reduction step is possible. A state is defined just if it reduces (in a finite number of steps) to a value state. Using these basic notions we define the operational approximation and equivalence relations in the usual way in terms of definedness in all program contexts. This is the basic semantic framework, independent of the choice of primitive operations. Within this framework we define the notion of uniform semantics and develop tools for proving laws of approximation and equivalence in λ-languages with uniform semantics. For a particular choice of operations what remains is to define the structure of state contexts, and provide the reduction rules for each primitive operation.

The uniformity requirements are that each reduction rule hold not only for traditional expressions, but also for expressions containing parameters or meta variables. This parametric notion of computation is best presented using the idea of a context and the treatment here follows the general theory presented in [27]. Contexts can be formalized by adding meta-variables or parameters to the syntax of expressions. The novelty of our approach is that we decorate parameters with information recording pending substitutions which are to be applied when the parameter is instantiated. This has several desirable consequences. One is that alpha conversion is valid for contexts, unlike the traditional lambda-calculus contexts. The other is that symbolic execution commutes with parameter instantiation, which simplifies reasoning about program equivalence.

2.1. Expression Syntax of a λ-language. Fix two disjoint countably infinite sets, $\mathbf{X}$, of variables, and $\mathbf{P}$ of parameters. The basic syntax of a λ-language is then determined by specifying three sets:

($\mathbf{A}$): a countable set of atoms, $\mathbf{A}$, disjoint from $\mathbf{X}$ and $\mathbf{P}$;
($\mathbf{O}$): a family of operation symbols $\mathbf{O} = \{\mathbf{O}_n \mid n \in \mathbf{N}\}$ ($\mathbf{O}_n$ is a set of n-ary operation symbols) disjoint from $\mathbf{X} \cup \mathbf{A} \cup \mathbf{P}$; and
($\mathbf{V}$): the set of value expressions, a subset of expressions, $\mathbf{V}$, that we will specify in more detail immediately after the definition of the syntax.

We assume that $\mathbf{O}$ contains at least the binary operation $\mathtt{app}$ (lambda application). As we shall see later by taking $\mathbf{A} = \{\ \}$ and $\mathbf{O} = \{\mathtt{app}\}$ we obtain the expressions of the pure call-by-value lambda calculus, Λ_{v}.

Definition 2.1 ($\mathbf{E}$, $\mathbf{L}$, $\mathbf{S}$, $\mathbf{F}$)**:** The set of expressions, $\mathbf{E}$, and the set of λ-abstractions, $\mathbf{L}$, the set of value substitutions, $\mathbf{S}$, and the set of parameter substitutions (fillings), $\mathbf{F}$ are defined as the least sets satisfying the following equations:

$$\mathbf{E} = \mathbf{X} \cup \mathbf{P}^{\mathbf{S}} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{O}_n(\mathbf{E}^n)$$

$$\mathbf{L} = \lambda\mathbf{X}.\mathbf{E}$$

$$\mathbf{S} = \mathrm{Fmap}[\mathbf{X}, \mathbf{V}]$$

$$\mathbf{F} = \mathrm{Fmap}[\mathbf{P}, \mathbf{E}]$$

We let a range over $\mathbf{A}$, x, y, z range over $\mathbf{X}$, X, Y, Z range over $\mathbf{P}$, e range over $\mathbf{E}$, φ range over $\mathbf{L}$, σ range over $\mathbf{S}$, and Σ range over $\mathbf{F}$.

$\mathbf{P}^{\mathbf{S}}$ is the set of parameters, annotated or decorated by value substitutions. Value substitutions, $\mathbf{S}$, are finite maps from variables to value expressions. The domain of a substitution is written as $\mathrm{Dom}(\sigma)$, and is defined in the usual way. Parameter substitutions are finite maps from parameters to expressions. We write $\{x_i \mapsto v_i \mid i < n\}$ for the value substitution, σ, with domain $\{x_i \mid i < n\}$ such that $\sigma(x_i) = v_i$ for $i < n$. Similarly we write $\{X_i \mapsto e_i \mid i < n\}$ for the parameter substitution, Σ, with domain $\{X_i \mid i < n\}$ such that $\Sigma(X_i) = e_i$ for $i < n$. Note that a parameter decorated by a value substitution is an expression. This allows us to compute parametrically with partially specified expressions, and thus our expressions generalize the usual notion of context. In this more general setting we must be somewhat more careful to define certain basic notions.

Definition 2.2 ($\mathrm{FV}(e)$, $\mathrm{P}(e)$)**:** The free variables, $\mathrm{FV}(e)$, and the parameters, $\mathrm{P}(e)$, (which are always free) of an expression e are defined inductively. The novel clauses are

$$\mathrm{FV}(X^\sigma) = \bigcup_{x\in\mathrm{Dom}(\sigma)} \mathrm{FV}(\sigma(x)) \qquad \mathrm{P}(X^\sigma) = \bigcup_{x\in\mathrm{Dom}(\sigma)} \mathrm{P}(\sigma(x)) \cup \{X\}$$

We extend these to substitutions is the obvious fashion:

$$\mathrm{FV}(\Delta) = \bigcup_{\delta\in\mathrm{Dom}(\Delta)} \mathrm{FV}(\Delta(\delta)) \qquad \mathrm{P}(\Delta) = \bigcup_{\delta\in\mathrm{Dom}(\Delta)} \mathrm{P}(\Delta(\delta)) \qquad \text{for } \Delta \in \mathbf{S} \cup \mathbf{F}.$$

Definition 2.3 ($\mathbf{E_0}$, term, closed)**:** We adopt the convention that an expression with no parameters is called a *term*. The set of all terms is denoted by $\mathbf{E_0}$. Furthermore a *term* with no free variables is *closed*. Thus being closed implies having no parameters.

Definition 2.4 (e^σ, e^Σ)**:** e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$, taking care not to trap variables. Taking care not to trap variables amounts to defining simultaneous substitution into a lambda expression by the following scheme

$$(\lambda z.e)^\sigma = \lambda\nu.((e^{\{z\mapsto\nu\}})^\sigma) \qquad \text{for } \nu \text{ fresh, i.e. } \nu \notin \mathrm{FV}(e) \cup \mathrm{FV}(\sigma).$$

In the case of decorated parameters we define simultaneous substitution as follows, $(X^{\sigma_0})^\sigma = X^{(\sigma_0^\sigma)}$, where $\sigma_0^\sigma = \{x \mapsto \sigma_0(x)^\sigma \mid x \in \mathrm{Dom}(\sigma_0)\}$.

e^Σ is the result of simultaneous substitution of decorated occurrences of $X \in \mathrm{Dom}(\Sigma)$ in e by $\Sigma(X)$ instantiated by the (suitably substituted) decoration, again taking care not to trap variables (other than those in the range of the decoration). For decorated parameters it is defined as follows, $(X^\sigma)^\Sigma = \Sigma(X)^{\sigma^\Sigma}$ if $X \in \mathrm{Dom}(\Sigma)$, and X^{σ^Σ} otherwise, where σ^Σ is defined point-wise: $\sigma^\Sigma = \{x \mapsto \sigma(x)^\Sigma \mid x \in \mathrm{Dom}(\sigma)\}$. In the case of λ-abstractions, we define parameter substitution exactly as we would value substitution:

$$(\lambda x.e)^\Sigma = \lambda\nu.((e^{\{x\mapsto\nu\}})^\Sigma) \qquad \text{for } \nu \text{ fresh, i.e. } \nu \notin \mathrm{FV}(e) \cup \mathrm{FV}(\Sigma).$$

Example 2.5: Suppose that $\vartheta \in \mathbf{O}_2$. Let e be the expression $\lambda z.(X^{\{w\mapsto\vartheta(z,w)\}})$. Then (assuming distinct variable names name distinct variables):

(i) $e^{\{X\mapsto z\}}$ is $\lambda\nu.z$ since trapping is made explicit by the annotating substitution, not the surrounding context.
(ii) $e^{\{X\mapsto w\}}$ is $\lambda z.\vartheta(z,w)$ since w is trapped by the substitution annotating X.

(iii) $e^{\{X\mapsto y\}}$ is $\lambda z.y$ since y is neither trapped, nor forces us to α-convert z.
(iv) $e^{\{w\mapsto y\}}$ is $\lambda z.(X^{\{w\mapsto\vartheta(z,y)\}})$, since the only free occurrence of w is in the range of the substitution annotating X.
(v) $e^{\{w\mapsto z\}}$ is $\lambda\nu.(X^{\{w\mapsto\vartheta(\nu,z)\}})$ since substitution is defined to avoid trapping.
(vi) $e^{\{w\mapsto w\}}$ is $\lambda z.(X^{\{w\mapsto\vartheta(z,w)\}})$ since no trapping takes place.

The notion of being equivalent modulo the renaming of bound variables easily extends to this more general setting [27] by simply clarifying what can and cannot be bound: parameters are *never* bound; variables in the domain of an annotating substitution are *never* bound; variables in the range of an annotating substitution *may* be bound. Using this one can generate the α-equivalence relation, $\overset{\alpha}{\equiv}$, by a set of rules, the new one being:

$$\frac{\sigma_0(x) \overset{\alpha}{\equiv} \sigma_1(x) \qquad \text{for every } x \in \mathrm{Dom}(\sigma_i)}{X^{\sigma_0} \overset{\alpha}{\equiv} X^{\sigma_1}} \qquad \text{provided } \mathrm{Dom}(\sigma_0) = \mathrm{Dom}(\sigma_1)$$

One last piece of notation concerning annotated parameters appearing in expressions. The set of *trapped* variables, $\mathrm{Traps}(e)$, is defined to be the smallest set of variables that contains the domains of any substitution that annotates an occurrence of a parameter in e. On the other hand the *domain* of e, $\mathrm{Dom}(e)$, is defined to be the largest set of variables contained in the domain of every substitution that annotates an occurrence of a parameter in e.

Definition 2.6 ($\mathrm{Traps}(e)$, $\mathrm{Dom}(e)$)**:** These amount to a simple inductive definitions, the interesting clauses being:

$$\mathrm{Traps}(X^\sigma) = \bigcup_{x\in\mathrm{Dom}(\sigma)} \mathrm{Traps}(\sigma(x)) \cup \mathrm{Dom}(\sigma)$$

$$\mathrm{Dom}(X^\sigma) = \bigcap_{x\in\{y\in\mathrm{Dom}(\sigma) \,\big|\, \mathrm{P}(\sigma(y))\neq\emptyset\}} \mathrm{Dom}(\sigma(x)) \cap \mathrm{Dom}(\sigma)$$

The domain of an expression is useful in expressing certain necessary closure requirements, as suggested by the following lemma.

Lemma 2.7 ($\mathrm{Dom}(e)$)**:** Suppose $\mathrm{FV}(e) = \emptyset$, and $\mathrm{FV}(\Sigma) \subseteq \mathrm{Dom}(e)$. Then $\mathrm{FV}(e^\Sigma) = \emptyset$

Definition 2.8 (Value Expressions ($\mathbf{V}$))**:** The set of value expressions, $\mathbf{V}$, contains all variables, atoms, and lambdas. It may in addition contain expressions of the form $\vartheta(v^n)$. $\mathbf{V}$ must also satisfy:

(triv) $\quad \mathbf{V}$ is closed under $\overset{\alpha}{\equiv}$

(vsub) $\quad v \in \mathbf{V} \Rightarrow v^\sigma \in \mathbf{V}$

(inst) $\quad v \in \mathbf{V} \Rightarrow v^\Sigma \in \mathbf{V}$

(dich) $\quad e^\Sigma \in \mathbf{V} \Rightarrow (e \in \mathbf{V}) \vee (e = X^\sigma \wedge \Sigma(X) \in \mathbf{V})$

We let v range over $\mathbf{V}$.

Operators, ϑ, that produce value expressions, are called constructors. In the languages considered here the binary pairing operation, `pr`, will serve as the prototypical constructor. The following lemma simply points out a simple consequence of the closure conditions on values.

Lemma 2.9 (inv)**:** $e^\sigma \in \mathbf{V} \;\Rightarrow\; e \in \mathbf{V}$

Proof: Pick e_0 such that $e_0^\sigma \in \mathbf{V}$, and let X be a fresh parameter. Put $e = X^\sigma$, $\Sigma = \{X \mapsto e_0\}$. Then

$$e^\Sigma = (X^\sigma)^\Sigma = \Sigma(X)^{(\sigma^\Sigma)} = \Sigma(X)^\sigma = e_0^\sigma \in \mathbf{V}$$

since $X^\sigma \notin \mathbf{V}$ we can use (**dich**) to conclude that $e_0 \in \mathbf{V}$. □

2.2. Operational Semantics. In a Landinesque language computation state is represented as a class of expressions. Each particular language will possess it's own class of state expressions, reflecting the nature of the primitive operations that it is based on. In what follows we fix a distinguished parameter $\circ$ to designate the position at which effects are to be observed in a context representing a computation state. We call this the *state* parameter.

Definition 2.10 (State expressions ($\mathbf{Z}$))**:** For a particular Landinesque language $\mathbf{Z}$ is assumed to be a subset of $\mathbf{E}$. We call $\mathbf{Z}$ the set of state expressions. $\mathbf{Z}$ is assumed to satisfy the following uniformity conditions:

(triv) $\mathbf{Z}$ is closed under $\overset{\alpha}{\equiv}$

(par) $\zeta \in \mathbf{Z} \Rightarrow \circ \in \mathrm{P}(\zeta)$

(vsub) $\zeta \in \mathbf{Z} \Rightarrow \zeta^\sigma \in \mathbf{Z}$ assuming $\circ \notin \mathrm{P}(\sigma)$

(inst) $\zeta \in \mathbf{Z} \Rightarrow \zeta^\Sigma \in \mathbf{Z}$ assuming $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{P}(\Sigma))$

ζ ranges over $\mathbf{Z}$ and $\circ \in \mathbf{Z}$ is the empty state expression.

Definition 2.11 (Values in a State ($\mathbf{V}_\zeta$))**:** The set of *values defined in a state* ζ is $\mathbf{V}_\zeta$:

$$\mathbf{V}_\zeta = \{v \in \mathbf{V} \mid \mathrm{FV}(v) \subseteq \mathrm{Dom}(\zeta)\}$$

Definition 2.12 (Computation States ($\mathbf{CS}$))**:** $\mathbf{CS} \overset{\triangle}{=} \mathbf{Z} : \mathbf{E}$ is the set of computation states. We let S range over $\mathbf{CS}$ and let $\zeta : e$ be the state with state context ζ and expression e. The computation state $\zeta : e$ is said to be a *value state* iff $e \in \mathbf{V}$. Given a state $\zeta : e$, we associate a corresponding expression by filling the state parameter in the context with the expression, i.e. $\zeta^{\{\circ \mapsto e\}}$. A state is *closed* just if its corresponding expression is closed, in other words if $\zeta^{\{\circ \mapsto e\}}$ has no free parameters or variables. Application of value and parameter substitutions to states is defined by in the obvious way: $(\zeta : e)^\sigma = \zeta^\sigma : e^\sigma$ and $(\zeta : e)^\Sigma = \zeta^\Sigma : e^\Sigma$. Note that by (**vsub**) (**inst**) these are only meaningful if $\circ \notin \mathrm{Dom}(\sigma)$ and $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{P}(\Sigma))$.

Definition 2.13 (Reduction ($\longrightarrow$, $\longrightarrow\!\!\!\!\!\succ\!\!\succ$) and Definedness ($\downarrow$))**:** Given a reduction relation for a Landinesque language: $\zeta : e \longrightarrow \zeta' : e'$, the following definitions are standard. The *transitive* closure of $\longrightarrow$ is $\longrightarrow\!\!\!\!\!\succ\!\!\succ$

Definedness:

$$(\zeta : e)\downarrow \;\Leftrightarrow\; \zeta : e \longrightarrow\!\!\!\succ \zeta' : v$$

Approximation:

$$(\zeta_0 : e_0) \preceq (\zeta_1 : e_1) \;\Leftrightarrow\; (\zeta_0 : e_0)\downarrow \;\Rightarrow\; (\zeta_1 : e_1)\downarrow$$

Equidefined:

$$(\zeta_0 : e_0) \updownarrow (\zeta_1 : e_1) \;\Leftrightarrow\; ((\zeta_0 : e_0)\downarrow \;\Leftrightarrow\; (\zeta_1 : e_1)\downarrow)$$

Equal:

$$(\zeta_0 : e_0) = (\zeta_1 : e_1) \;\Leftrightarrow\; (\exists v \in \mathbf{V}, \zeta \in \mathbf{Z})(\bigwedge_{j<2} (\zeta_i : e_i) \longrightarrow\!\!\!\succ \zeta : v)$$

Equivalued:

$$(\zeta_0 : e_0) \sim (\zeta_1 : e_1) \;\Leftrightarrow\; (\zeta_0 : e_0) \updownarrow (\zeta_1 : e_1) \wedge (\zeta_0 : e_0)\downarrow \;\Rightarrow\; (\zeta_0 : e_0) = (\zeta_1 : e_1)$$

Length:
$|\zeta : e|$ is the least $n \in \mathbf{N}$ such that $\zeta : e$ reduces to a value state in n steps, if $\zeta : e\downarrow$.

To define reduction rules for general Landinesque languages, and formulate the central properties of reduction and equivalence, we introduce the notions of redex and reduction context. Since evaluation is call-by-value, a redex is simply an non-constructor operator applied to the appropriate number of value expressions. Redexes and value expressions must be disjoint, thus we must account for the fact that some expressions of the form $\vartheta(v_1, \ldots, v_n)$ may be value expressions.

Definition 2.14 (Redexes ($\mathbf{E}_\mathrm{r}$))**:** The set of redexes, $\mathbf{E}_\mathrm{r}$, is defined by:

$$\mathbf{E}_\mathrm{r} = \mathbf{O}_n(\mathbf{V}^n) - \mathbf{V}$$

Note that redexes in our framework may or may not reduce. The point is that they are simply expressions of a particular shape, in other words: candidates for reduction. We use the distinguished parameter $\bullet$ to denote the *evaluation* parameter (or hole), and we define the notion of a reduction context, R, accordingly. Reduction contexts (also called evaluation contexts in the literature) identify the subexpression of an expression in which reduction to a value must occur next. They themselves represent the remainder of the computation, i.e the *continuation*. In our approach they correspond to the left-first, call-by-value reduction strategy of [35] and were first introduced by [13].

Definition 2.15 (Reduction Contexts ($\mathbf{R}$))**:** The set of reduction contexts, $\mathbf{R}$, is the subset of $\mathbf{E}$ defined by

$$\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{m+n+1}(\{v \in \mathbf{V} \mid \bullet \notin \mathrm{P}(v)\}^m, \mathbf{R}, \{e \in \mathbf{E} \mid \bullet \notin \mathrm{P}(e)\}^n)$$

We let R range over $\mathbf{R}$. We adopt the convention of writing $R[\![e]\!]$ instead of $R^{\{\bullet \mapsto e_1\}}$.

Observe that both the definition of redex, and the definition of reduction contexts depend on the particular choice of values, and thus vary from one Landinesque language to another. Also note that $\mathbf{R}$ will satisfy a similar set of uniformity conditions as those satisfied by states (Definition 2.10):

Lemma 2.16 (Rcx Uniformity)**:** Reduction contexts satisfy the following uniformity conditions:

(triv)	$\mathbf{R}$ is closed under $\overset{\alpha}{\equiv}$	
(par)	$R \in \mathbf{R} \Rightarrow \bullet \in \mathrm{P}(R)$	
(vsub)	$R \in \mathbf{R} \Rightarrow R^{\sigma} \in \mathbf{R}$	assuming $\bullet \notin \mathrm{P}(\sigma)$
(inst)	$R \in \mathbf{R} \Rightarrow R^{\Sigma} \in \mathbf{R}$	assuming $\bullet \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{P}(\Sigma))$

It is easy to check that an expression without parameters is either a value expression or decomposes uniquely into a redex placed in a reduction context (a proof can be found in [29]). This generalizes to the present situation in the following fashion.

Lemma 2.17 (Decomposition)**:** For any Landinesque language, if $e \in \mathbf{E}$ then either $e \in \mathbf{V}$ or e can be written uniquely as either

(i) $R[\,r\,]$ where R is a reduction context and $r \in \mathbf{E}_{\mathrm{r}}$, or else
(ii) $R[\,X^{\sigma}\,]$ where R is a reduction context, and $X^{\sigma} \in \mathbf{P}^{\mathbf{S}}$ is a decorated parameter.

In the latter case we say that the expression is *touching* the parameter, while in the former we say that the expression *may be reducible*. The requirement that the evaluation parameter does not occur in either the leading value expressions, or the trailing expressions is necessary for the uniquenss aspect of this lemma. A simple counterexample, due to Jonathan Ford, is the following:

$$R_0 = \vartheta(\bullet, \bullet) \quad R_1 = \vartheta(\bullet, r) \quad R_0[\,r\,] = \vartheta(r, r) = R_1[\,r\,].$$

For the languages considered here, the single step reduction relation is defined by giving a reduction rule for each operation. These rules are of the form:

$$\zeta_0 : R[\,\vartheta(v_1, \ldots, v_n)\,] \longrightarrow \zeta_1 : e.$$

In particular the rule for the `app` operation is the standard reduction rule for the call-by-value lambda calculus.

Definition 2.18 (Beta value reduction (β_{v}))**:** The reduction rule for `app` is the beta value rule:

$$(\beta_{\mathrm{v}})\ \ \zeta : R[\,\mathtt{app}(\lambda x.e, v)\,] \longrightarrow \zeta : R[\,e^{\{x \mapsto v\}}\,].$$

Notice that this rule is schematic in the actual states, reduction contexts, and values of a particular Landinesque language.

2.3. Uniform Semantics. We now specify what we mean by a λ-language having *uniform semantics*. The key requirement is that reduction steps that do not touch a parameter are uniformly independent of what the parameter might stand for. In addition, we require that: single step reduction is essentially deterministic; reduction is preserved by value substitution; a state, and its associated expression started in the empty state context, are equi-defined; and if one state reduces to another then the two states are equi-defined and the reduct has shorter computation length, if defined.

Definition 2.19 (Uniformity (**U**))**:** A λ-language is said to have *uniform semantics* if it satisfies the following:

(**i**) Functional modulo $\overset{\alpha}{\equiv}$ and implicit bindings:

$$(\bigwedge_{j<2}(\zeta_i : e_i) \overset{\alpha}{\equiv} (\zeta'_i : e'_i)) \Rightarrow ((\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \Rightarrow (\zeta'_0 : e'_0 \longrightarrow \zeta'_1 : e'_1))$$

$$(\bigwedge_{j<2} \zeta : e \longrightarrow \zeta_i : e_i) \Rightarrow \zeta_0^{\{\circ\mapsto e_0\}} \overset{\alpha}{\equiv} \zeta_1^{\{\circ\mapsto e_1\}}.$$

(**ii**) Uniform in value substitutions:

$$\zeta : e \longrightarrow \zeta' : e' \Rightarrow (\zeta : e)^{\sigma} \longrightarrow (\zeta' : e')^{\sigma}$$

provided $\mathrm{Dom}(\sigma) \cap (\mathrm{Traps}(\zeta) \cup \mathrm{Traps}(\zeta')) = \emptyset$ and $\circ \notin \mathrm{P}(\sigma)$.

(**iii**) State evaluation:

$$\zeta : e \updownarrow \circ : \zeta^{\{\circ\mapsto e\}}$$

(**iv**) Well-founded:

$$(\zeta : e \longrightarrow \zeta' : e' \wedge \zeta : e\downarrow) \Rightarrow (\zeta' : e'\downarrow \wedge |\zeta' : e'| < |\zeta' : e'|)$$

(**v**) Parametric:

$$\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1 \Rightarrow (\zeta_0 : e_0)^{\Sigma} \longrightarrow (\zeta_1 : e_1)^{\Sigma}$$

for any $\Sigma \in \mathbf{F}$ with $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{P}(\Sigma))$.

(**vi**) Dichotomy (Either computation touches a parameter or is parametric): Assuming $\circ \notin \mathrm{Dom}(\Sigma)$, if $(\zeta : e)^{\Sigma} \longrightarrow \zeta' : e'$ then either

$\zeta : e$ touches a parameter in the domain of Σ, or

$\zeta : e \longrightarrow \zeta_1 : e_1$, for some $\zeta_1 : e_1$ such that $\zeta' : e' \overset{\alpha}{\equiv} (\zeta_1 : e_1)^{\Sigma}$.

(**vii**) Closure:

(i) $\quad (\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \Rightarrow \mathrm{FV}(\zeta_1^{\{\circ\mapsto e_1\}}) \subseteq \mathrm{FV}(\zeta_0^{\{\circ\mapsto e_0\}})$

(ii) $\quad (\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \Rightarrow \mathrm{P}(\zeta_1 : e_1) \subseteq \mathrm{P}(\zeta_0 : e_0) \wedge (\circ \notin \mathrm{P}(e_0) \Rightarrow \circ \notin \mathrm{P}(e_1))$

In the languages we consider (**U**) holds for the following reasons. (**U.i**) holds because the only non-determinism in a reduction step is the choice of names used in the state context. (**U.ii**) holds because reductions that do not depend on the values of free variables, are parametric in the values that those variables take. (**U.iii**) holds because reduction of $\circ : \zeta^{\{\circ\mapsto e\}}$ essentially recreates the state context ζ. (**U.iv**) follows since if a state is defined, then any reduction makes progress. Clearly if the reduct state is defined, then the original state is defined. (**U.v**) and (**U.vi**) formalizes the uniformity requirement for reduction steps. These are satisfied by reduction rules that treat the reduction context as an abstract entity, and that depend on the kind of construction of a redex argument, but not on any information about subparts. This is easily expressed using the parameters. Finally, (**U.vii**) holds because computation neither introduces new parameters, nor new free variables.

2.4. Approximation and Equivalence. Now we define operational approximation and equivalence on terms and lay the ground work for studying properties of these relations. In what follows we fix a particular distinguished parameter, X, distinct from $\circ$ and $\bullet$. We let C range over expressions with X as the only free parameter, with the added condition that the substitutions annotating X be restricted to renamings (finite maps from variables to variables). Such expressions play the role of traditional lambda-calculus contexts, and we extend our convention, stated in definition 2.15, of sometimes writing $C[\![e]\!]$ instead of $C^{\{X \mapsto e\}}$. Note however that for example the traditional context $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}([\,], z))$ does not correspond to $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}(X, z))$ but rather to one where the trappings have been made explicit at the occurrence of the X parameter: $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}(X^{\{x \mapsto x, z \mapsto z\}}, z))$.

For each traditional context C there is a corresponding expression, $\hat{C}$ such that

$$C[\![e]\!] = \hat{C}^{\{X \mapsto e\}}.$$

$\hat{C}$ is obtained by replacing each occurrence of $[]$ in C by the parameter X, and decorating each such occurrence of X with a binding substitution $\{x_i \mapsto x_i \mid x_i \in T\}$ where T is the set of lambda variables having the hole occurrence in their scope. For example, the expression corresponding to $\lambda x.\bullet$ is $\lambda x.X^{\{x \mapsto x\}}$. This guarantees that the following definition corresponds to the traditional notion of operational equivalence.

Definition 2.20 (Approximation $e_0 \sqsubseteq e_1$, Equivalence $e_0 \cong e_1$)**:** For terms e_0, e_1 define

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall C \mid C[\![e_0]\!], C[\![e_1]\!]\text{closed})(\circ : C[\![e_0]\!] \;\preceq\; \circ : C[\![e_1]\!])$$

$$e_0 \cong e_1 \;\Leftrightarrow\; e_0 \sqsubseteq e_1 \,\wedge\, e_1 \sqsubseteq e_0$$

Note that we are restricting our attention to terms, rather than arbitrary expressions. It is easy to see that operational approximation is a congruence on terms: if $e_0 \sqsubseteq e_1$, then $C[\![e_0]\!] \sqsubseteq C[\![e_1]\!]$. Similarly for operational equivalence.
We may now state the main result concerning Landinesque languages with uniform semantics, the CIU theorem.

Theorem 2.21 (CIU)**:** For a λ-language with uniform semantics:

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall \zeta, R, \sigma \mid \bigwedge_{j<2} \zeta : R[\![e_j^\sigma]\!] \;\text{closed})(\zeta : R[\![e_0^\sigma]\!] \;\preceq\; \zeta : R[\![e_1^\sigma]\!])$$

There are several proofs of this result in the literature. The theorem first appeared in [29] and the proof (sketch) presented there used techniques similar to those developed here. A somewhat more detailed and general version of this same technique appeared in [42]. Recently this same proof has been verified using the PVS theorem prover [15]. A second, distinct, proof was presented in [22] that simply shows that the CIU relation is a congruence.

Corollary 2.22 (Substitutivity)**:** In a λ-language with uniform semantics operational approximation is preserved by substitution: if $e_0 \sqsubseteq e_1$, then $e_0^{\sigma} \sqsubseteq e_1^{\sigma}$. Similarly for operational equivalence.

Example 2.23 (Call-by-value λ-calculus (Λ_{v}))**:** As a first example we complete the definition of the language Λ_{v}. Λ_{v} has no atoms ($\mathbf{A} = \{\ \}$) and a single operation, $\mathbf{O} = \{\mathtt{app}\}$. There is one state, the empty state, represented by the state parameter $\circ$. In Λ_{v} (as in all Landinesque languages) the reduction rule for `app` is the β_{v} rule (Definition 2.18).

It is easy to see that Λ_{v} satisfies the uniformity conditions (Definition 2.19). The only hard part is to show that value substitution and β_{v} reduction commute, which is a standard result.

2.5. Context Independent Reduction. To state the theorems underlying computational equivalences in this general setting we introduce the notion of a context independent (CI) redex.

Definition 2.24 (Context independent reduction ($\xrightarrow{\mathrm{ci}}\!\!\succ$))**:** A redex $\vartheta(v_1, \ldots, v_n)$ is CI if the interpretation is independent of the context: either there is no reduction possible in any computation context (state and reduction context), or the redex is replaced by the same reduct expression in any computation context. That is, exactly one of the following holds:

1. For any ζ, R there is no $\zeta' : e'$ such that $\zeta : R[\![\vartheta(v_1, \ldots, v_n)]\!] \longrightarrow \zeta' : e'$.
2. There is some e such that $\zeta : R[\![\vartheta(v_1, \ldots, v_n)]\!] \longrightarrow \zeta : R[\![e]\!]$ for any ζ, R.

A CI redex neither examines nor modifies its context (state or reduction). For example, any redex with operator `app` is CI.

We write $e \xrightarrow{\mathrm{ci}}\!\!\succ e'$ if $\zeta : R[\![e]\!] \longrightarrow\!\!\succ \zeta : R[\![e']\!]$ by a sequence of CI steps for any ζ and R.

The intuition that two expressions are equivalent if they have a common reduct, is justified by Theorem 2.25 below. Similarly reasoning that two expressions are equivalent if they result from placing a third expression in computationally equivalent reduction contexts is justified by Theorem 2.26.

Theorem 2.25 (Equi-reduct)**:** In a λ-language with uniform semantics, if there is some e such that $e_j \xrightarrow{\mathrm{ci}}\!\!\succ e$ for $j < 2$ then $e_0 \cong e_1$.

Proof: Assume $e_j \xrightarrow{\mathrm{ci}}\!\!\succ e$ for $j < 2$. By CIU (and symmetry), to show that $e_0 \cong e_1$ we need only show $\zeta : R[\![e_0^{\sigma}]\!] \preceq \zeta : R[\![e_1^{\sigma}]\!]$ for all closing ζ, R, and σ. Choose any such ζ, R, σ and assume $\zeta : R[\![e_0^{\sigma}]\!]\downarrow$, furthermore we may choose ζ and σ such that $\mathrm{Dom}(\sigma) \cap \mathrm{Traps}(\zeta) = \emptyset$. By assumption pick e such that

$$\zeta : R[\![e_j]\!] \longrightarrow\!\!\succ \zeta : R[\![e]\!] \qquad \text{for any } \zeta \text{ and } R.$$

Thus by (**U.2**) we have

$$(\zeta : R[\![e_j]\!])^{\sigma} \longrightarrow\!\!\succ (\zeta : R[\![e]\!])^{\sigma} \qquad \text{for the particular } \zeta \text{ and } R \text{ chosen above.}$$

Consequently

$\zeta : R[\,e_j^\sigma\,] \longrightarrow\!\!\!\!\rightarrowtail \zeta : R[\,e^\sigma\,]$ since $\zeta = \zeta^\sigma$ and $R = R^\sigma$.

Thus, by (**U.4**), $\zeta : R[\,e^\sigma\,]\downarrow$, and hence $\zeta : R[\,e_1^\sigma\,]\downarrow$. □

Theorem 2.26 (Equi-rcx)**:** In a λ-language with uniform semantics, if for z fresh there are $e_0 \cong e_1$ such that $R_j[\,z\,] \xrightarrow{\text{ci}}\!\!\!\!\rightarrowtail e_j$ for $j < 2$, then $R_0[\,e\,] \cong R_1[\,e\,]$ for any e.

To provide simple instructive examples of the use of these two principles we will introduce the ubiquitous `let` construct that will feature in the subsequent development, and establish its more well known properties.

Definition 2.27 (`let`)**:** The `let` form is very common programming language construct abbreviating λ-application:

$$\mathtt{let}\{x := v\}e \overset{\triangle}{=} \mathtt{app}(\lambda x.e, v).$$

Example 2.28 (`let`)**:** As mentioned above β_{v} reduction of $\mathtt{app}(\lambda x.e, v)$ is CI, thus an immediate consequence of Theorem 2.25 is that

(let.i) $\mathtt{let}\{x := v\}e \cong e^{\{x\mapsto v\}}$.

Assume that $x \notin \text{FV}(R_0)$ and $R_1 = \mathtt{let}\,\{x := \bullet\}R_0[\,x\,]$. Then for z fresh:

$$R_1[\,z\,] \xrightarrow{\text{ci}}\!\!\!\!\rightarrowtail R_0[\,x\,]^{\{x\mapsto z\}} = R_0[\,z\,]$$

So by Theorem 2.26:

(let.ii) $R_0[\,e\,] \cong \mathtt{let}\,\{x := e\}R_0[\,x\,]$

Another example is obtained by taking $R_0 = \mathtt{let}\,\{x := \bullet\}\,\mathtt{let}\,\{y := e_1\}e$ and $R_1 = \mathtt{let}\,\{y := \mathtt{let}\,\{x := \bullet\}e_1\}e$ with the added proviso that x and y are distinct, and $x \notin \text{FV}(e)$. Then for z fresh:

$$R_0[\,z\,] = \mathtt{let}\,\{x := z\}\,\mathtt{let}\,\{y := e_1\}e$$

$$\xrightarrow{\text{ci}}\!\!\!\!\rightarrowtail (\mathtt{let}\,\{y := e_1\}e)^{\{x\mapsto z\}} = \mathtt{let}\,\{y := e_1^{\{x\mapsto z\}}\}e$$

$$R_1[\,z\,] = \mathtt{let}\,\{y := \mathtt{let}\,\{x := z\}e_1\}e$$

$$\xrightarrow{\text{ci}}\!\!\!\!\rightarrowtail \mathtt{let}\,\{y := e_1^{\{x\mapsto z\}}\}e$$

Thus again by Theorem 2.26:

(let.iii) $\mathtt{let}\,\{x := e_0\}\,\mathtt{let}\,\{y := e_1\}e \simeq \mathtt{let}\,\{y := \mathtt{let}\,\{x := e_0\}e_1\}e$

These three principles hold in any uniform Landinesque language. They are fundamental in that they are equivalent to the let-rules of the computational Λ_{c} calculus [33].

An even stronger, but nevertheless useful notion, is being *ineffectual* which captures the idea of *single CI stepping to a value*. In that every reduction rule for the operation leaves its context (state and reduction) unchanged and produces a value.

Definition 2.29 (Ineffectual)**:** An operation $\vartheta \in \mathbf{O}$ is said to be *ineffectual*, iff every reduction rule for the operation leaves its context (state and reduction) unchanged and produces a value, i.e is of the form:

$$\zeta : R[\![\vartheta(v_1, \ldots, v_n)]\!] \longrightarrow \zeta : R[\![v]\!].$$

An operation that is not ineffectual is said to have *effects*.

Of course the interesting operations are not the ineffectual ones, we simply use ineffectualness to separate out the uninteresting operations, since they have many more properties than the ones with effects. The only operation common to all Landinesque languages is `app` and it has effects.

§3. The Syntax and Semantics of Formulas.

3.1. Syntax. The first order fragment of our logic is a minor generalization of classical first order logic. The atomic formulas, $\mathbf{W}_0$, assert the definedness, equivaluedness and operational equivalence of terms, $\mathbf{E_0}$, i.e. parameter-free expressions.

In addition to the usual first-order formula constructions, we add a `let`-assertion:

$$\texttt{let}\,\{x := e\}\Phi$$

where Φ is a formula, x a variable, and e a term. The formula $\texttt{let}\,\{x := e\}\Phi$ asserts that if execution of e terminates with value v, then Φ holds in the resulting state, with x bound to the value v. This provides a means for expressing the effects of evaluating an expression. An instructive example is the formula

(newness) $\quad (\forall \bar{z})(\forall x)\;\texttt{let}\,\{y := \vartheta(\bar{z})\}\neg(x \sim y)$

expresses that ϑ creates *new* values, distinct from any values existing prior to evaluation, such operations are usually called constructors, or allocators. Note that the `let`-assertion is a binding operator akin to $\forall$.

Definition 3.1 ($\mathbf{W}$)**:**

$$\mathbf{W}_0 = (\mathbf{E_0} \sim \mathbf{E_0}) \cup (\mathbf{E_0} \cong \mathbf{E_0})$$

$$\mathbf{W} = \mathbf{W}_0 \cup (\neg\mathbf{W}) \cup (\mathbf{W} \;\Rightarrow\; \mathbf{W}) \cup (\texttt{let}\,\{\mathbf{X} := \mathbf{E_0}\}\mathbf{W}) \cup (\forall\mathbf{X})(\mathbf{W})$$

3.2. Semantics. The meaning of formulas is given by a Tarskian satisfaction relation $\zeta \models \Phi[\sigma]$.

Definition 3.2 ($\zeta \models \Phi[\sigma]$)**:**
Assume ζ, σ, Φ, e_j satisfy $\mathrm{FV}(\Phi^\sigma), \mathrm{FV}(e_j^\sigma) \subseteq \mathrm{Dom}(\zeta)$ and $\mathrm{P}(e_j) = \emptyset$ for $j < 2$, and $\mathrm{FV}(\zeta) = \emptyset$. Then we define the satisfaction relation $\zeta \models \Phi[\sigma]$ by induction on

the structure of Φ:

$$\begin{array}{lll}
\zeta \models (e_0 \sim e_1)[\sigma] & \text{iff} & (\forall R \in \mathbf{R} \mid \mathrm{FV}(R) \subseteq \mathrm{Dom}(\zeta)) \\
& & \quad (\zeta : R[\![e_0^\sigma]\!] \sim \zeta : R[\![e_1^\sigma]\!]) \\
\zeta \models (e_0 \cong e_1)[\sigma] & \text{iff} & (\forall R \in \mathbf{R} \mid \mathrm{FV}(R) \subseteq \mathrm{Dom}(\zeta)) \\
& & \quad (\zeta[\![R[\![e_0^\sigma]\!]]\!] \updownarrow \zeta[\![R[\![e_1^\sigma]\!]]\!]) \\
\zeta \models \neg\Phi[\sigma] & \text{iff} & \text{not } (\zeta \models \Phi[\sigma]) \\
\zeta \models (\Phi_0 \Rightarrow \Phi_1)[\sigma] & \text{iff} & (\zeta \models \Phi_0[\sigma]) \quad \text{implies} \quad (\zeta \models \Phi_1[\sigma]) \\
\zeta \models \mathtt{let}\,\{x := e\}\Phi[\sigma] & \text{iff} & (\zeta : e^\sigma \longrightarrow\!\!\!\succ \zeta' : v) \quad \text{implies} \quad \zeta' \models \Phi[\sigma\{x := v\}]) \\
\zeta \models (\forall x)\Phi[\sigma] & \text{iff} & (\forall v \in \mathbf{V}_\zeta)(\zeta \models \Phi[\sigma\{x := v\}])
\end{array}$$

As is usual in logic we define the subsidiary notions of validity and logical consequence as follows:

$$\begin{array}{lll}
\models \Phi & \text{iff} & (\forall \zeta, \sigma \mid \mathrm{FV}(\Phi^\sigma) \subseteq \mathrm{Dom}(\zeta))(\zeta \models \Phi[\sigma]) \\
\Phi_0 \models \Phi_1 & \text{iff} & \models \Phi_0 \Rightarrow \Phi_1
\end{array}$$

Note that conjunction, $\wedge$, and disjunction, $\vee$ and the biconditional, $\Leftrightarrow$, are all definable in the usual manner. Also, termination and non-termination are simple abbreviations. We let $\downarrow e$ (termination) abbreviate $\neg(\mathtt{let}\,\{x := e\}\mathtt{False})$ and $\uparrow e$ (non-termination) abbreviate its negation $\mathtt{let}\,\{x := e\}\mathtt{False}$ where $\mathtt{False}$ is any unsatisfiable assertion, such as $\neg(x \sim x)$. Termination is one form of definedness. In fact, there is a plethora of notions of definedness that can be expressed. A stronger notion of definedness is that of being equivalent (either via $\sim$ or via $\cong$) to a value, for example an operation that satisfies the newness principle above will never be defined in this stronger sense.

§4. Syntax and Semantics of Classes.

Using methods of [5, 9] and [40], we extend our theory to include a general theory of classifications (classes for short). With the introduction of classes, principles such as structural induction, as well as principles accounting for the effects of an expression can easily be expressed.

4.1. Syntax of Classes. We extend the syntax to include class terms. Class terms are either class variables, $\mathbf{X}^c$, class constants, $\mathbf{A}^c$, or comprehension terms, $\{x \mid \Phi\}$.

Definition 4.1 (K): The set $\mathbf{K}$ of class terms is defined by

$$\mathbf{K} = \mathbf{X}^c \cup \mathbf{A}^c \cup \{\mathbf{X} \mid \mathbf{W}\}$$

We extend the atomic formulas to include class membership and the set of formulas $\mathbf{W}$ to include quantification over class variables. We should point out that $\mathbf{K}$ and $\mathbf{W}$ form a mutual recursive definition. The definition of expressions remains unchanged.

Definition 4.2 (W):

$$\mathbf{W}_0 = (\mathbf{E_0} \sim \mathbf{E_0}) \cup (\mathbf{E_0} \cong \mathbf{E_0}) \cup (\mathbf{E_0} \in \mathbf{K})$$

$$\mathbf{W} = \mathbf{W}_0 \cup \neg\mathbf{W} \cup (\mathbf{W} \Rightarrow \mathbf{W}) \cup (\mathtt{let}\,\{\mathbf{X} := \mathbf{E_0}\}\mathbf{W}) \cup (\forall\mathbf{X})(\mathbf{W}) \cup (\forall\mathbf{X}^c)\mathbf{W}$$

We let κ range over $\mathbf{X}^c$ and K range over $\mathbf{K}$. We will use identifiers beginning with an upper case letter in **This** font (for example **Val**) for class constants.

4.2. Semantics of Classes. To give semantics to the extended language, we extend the satisfaction relation as follows. Firstly we let $\mathbf{K}_\zeta$, the set of class values over ζ, be the set of subsets of $\mathbf{V}_\zeta$ (values over ζ) closed under $\cong$. Another possible choice for the set of class values is the set of definable sets, i.e. the set of class code extensions (cf. [7, 40]).

We extend value substitutions to map class variables to class values. This is used to define $[K]^\sigma_\zeta$, the value of a class term, K, relative to the given state, ζ, and the closing value substitution σ. In principle, the class term evaluation is relative to a valuation for class constants, but since all of our class constants are introduced by definitional extension, this can be ignored.

Definition 4.3 ($[\mathbf{K}]^\sigma_\zeta$)**:**

$$[\kappa]^\sigma_\zeta = \sigma(\kappa)$$

$$[\{x \mid \Phi\}]^\sigma_\zeta = \{v \in \mathbf{V}_\zeta \mid \zeta \models \Phi[\sigma\{x := v\}]\}$$

We then extend the satisfaction relation to formulas involving class terms and quantifiers.

Definition 4.4 ($\zeta \models \Phi[\sigma]$)**:** The new clauses in the inductive definition of satisfaction are:

$$\zeta \models e \in K[\sigma] \;\Leftrightarrow\; (\exists v \in \mathbf{V}_{\mathrm{Dom}(\zeta)})(\zeta : e^\sigma \longrightarrow\!\!\!\succ \zeta : v \,\wedge\, v \in [K]^\sigma_\zeta)$$

$$\zeta \models (\forall \kappa)\Phi[\sigma] \;\Leftrightarrow\; (\forall K \in \mathbf{K}_{\mathrm{Dom}(\zeta)})(\zeta \models \Phi[\sigma\{\kappa := K\}])$$

It is important to note that if $\zeta \models e \in K[\sigma]$, then e evaluates (in the appropriate state) to a value without altering that state.

Example 4.5 (Class definitions)**:** We define (extensional) equality and subset relations on classes in the usual manner.

$$K_0 \subseteq K_1 \;\stackrel{\triangle}{=}\; (\forall x)(x \in K_0 \;\Rightarrow\; x \in K_1)$$

$$K_0 \equiv K_1 \;\stackrel{\triangle}{=}\; K_0 \subseteq K_1 \,\wedge\, K_1 \subseteq K_0$$

The class $K_0 \to K_1$ is the set of lambdas that define total functions that have no effect on the state context.

$$K_0 \to K_1 \;\stackrel{\triangle}{=}\; \{f \mid (\forall x \in K_0)(\exists y \in K_1)\mathtt{app}(f, x) \sim y\}$$

§**5. Proof Theory.** Since contextual assertions are akin to modalities, we give a Hilbert style presentation. In the long run a natural deduction style system in the style of Prawitz [36] may be more desirable. Jacob Frost has developed a natural deduction presentation of this system [17, 18] for the $\lambda_{\mathtt{mk}}$ language [22], a Landinesque language with primitives for manipulating memory cells, and implemented it in the proof assistant *Isabelle*.

Definition 5.1 ($\vdash \Phi$)**:** The consequence relation, $\vdash$, is the smallest relation on $\mathbf{W}$ that is closed under the rules given below.

The rules are partitioned into several groups. Each group of rules is given a label, for future reference, and members of the group are numbered. For example (**E.i**) refers to the first rule in the group of equivalence and evaluation rules (the second group below). A rule has a (possibly empty) set of premises and a conclusion. In the case that the set of premises is non-empty the rule is displayed with a horizontal bar separating the premises from the conclusion.

Definition 5.2 ($\simeq$)**:** Most axioms hold true for both equivaluedness, $\sim$, and operational equivalence, $\cong$, If this is the case, then rather than write out the principle twice, we use the symbol $\simeq$ to range over these two equivalence relations.

One important reason for introducing $\sim$ is that important principles fail for $\cong$. In particular (**C.iii**) below fails as indicated in [30].

5.1. Basic Equivalence and Evaluation Rules. The first two sets of rules concerning equivaluedness hold true also of operational equivalence. They are equivalence relations, (**E.i, E.ii, E.iii**). They satisfy a certain restricted form of substitutivity, (**E.iv**).

They are also preserved under simple forms of evaluation, (**let.i, let.ii, let.iii**) that we treated in Example 2.28.

Equivalence and Evaluation Axioms (E).

(i) $\vdash e_0 \simeq e_0$

(ii) $\vdash (e_0 \simeq e_1 \wedge e_1 \simeq e_2) \Rightarrow e_0 \simeq e_2$

(iii) $\vdash e_0 \simeq e_1 \Rightarrow e_1 \simeq e_0$

(iv) $\vdash e_0 \simeq e_1 \Rightarrow \mathtt{let}\,\{x := e_0\}e \simeq \mathtt{let}\,\{x := e_1\}e$

Evaluation Axioms (let).

(i) $\vdash \mathtt{app}(\lambda x.e, v) \simeq e^{\{x:=v\}} \simeq \mathtt{let}\{x := v\}e$

(ii) $\vdash R[\,e\,] \simeq \mathtt{let}\{x := e\}R[\,x\,]$

(iii) $\vdash \mathtt{let}\,\{x := e_0\}\,\mathtt{let}\,\{y := e_1\}e \simeq \mathtt{let}\,\{y := \mathtt{let}\,\{x := e_0\}e_1\}e$

The principle (**let.ii**) is subject to the side condition that that $x \notin \mathrm{FV}(R)$, while in (**let.iii**) we require $x \notin \mathrm{FV}(e)$. The remaining axioms and rules concerning operational equivalence are: (**$\cong$.i**), equivaluedness implies operational equivalence; and (**$\cong$.ii**), operational equivalence is not the same as equivaluedness on abstractions.

Operational Equivalence Rules ($\cong$).

(i) $\vdash e_0 \sim e_1 \Rightarrow e_0 \cong e_1$

(ii) $$\frac{\vdash e_0 \cong e_1}{\vdash \lambda x.e_0 \cong \lambda x.e_1}$$ **(The ξ Rule)**

5.2. Contextual Rules. A contextual assertion is a modality and as such possesses a rule akin to necessitation, (**C.i**). Note that this is a rule of proof and not an implication. The remaining axioms concerning contextual assertions are: (**C.ii**), contextual assertions distribute across the equivalences, (the converse is false); (**C.iii**), a form of

contextual assertion introduction involving equivaluedness (the corresponding principle for operational equivalence is false); (**C.iv**), a principle akin to β conversion; and (**C.v**), a principle allowing for the manipulation of contexts.
Contextual Rules (C).

(i) $\dfrac{\vdash \Phi}{\vdash \mathtt{let}\,\{x := e\}\Phi}$ for any x, e, and Φ. **(Context Introduction)**

(ii) $\vdash \mathtt{let}\,\{x := e\}(e_0 \simeq e_1) \;\Rightarrow\; \mathtt{let}\,\{x := e\}e_0 \simeq \mathtt{let}\,\{x := e\}e_1$

(iii) $\vdash e_0 \sim e_1 \;\Rightarrow\; (\mathtt{let}\,\{x := e_0\}\Phi \;\Leftrightarrow\; \mathtt{let}\,\{x := e_1\}\Phi)$

(iv) $\vdash \mathtt{let}\,\{x := v\}\Phi \;\Leftrightarrow\; \Phi^{\{x:=v\}}$

(v) $\vdash \mathtt{let}\,\{x := e_0\}\,\mathtt{let}\,\{y := e_1\}\Phi \;\Leftrightarrow\; \mathtt{let}\,\{y := \mathtt{let}\,\{x := e_0\}e_1\}\Phi$

The principle (**C.v**) is subject to the condition that $x \notin \mathrm{FV}(\Phi)$.

Example 5.3: Note that using (**C.i**) in the special case that e is a value, allows us to also use (**C.iv**) to eliminate the `let` in preference for the substitution, thus the following is a derivable rule.

$\dfrac{\vdash \Phi}{\vdash \Phi^{\sigma}}$ **(Substitution)**

5.3. Logical Rules. The propositional rules are, in addition to the usual Hilbert style presentation of modus ponens, (**P.iii**), and a generating set of tautologies, (**P.i**) a modal axiom corresponding to **K** and its converse, (**P.ii**).
Propositional Rules (P).

(i) $\vdash \Phi$ provided Φ is an instance of a tautology

(ii) $\vdash \mathtt{let}\,\{x := e\}(\Phi_0 \;\Rightarrow\; \Phi_1) \;\Leftrightarrow\; (\mathtt{let}\,\{x := e\}\Phi_0 \;\Rightarrow\; \mathtt{let}\,\{x := e\}\Phi_1)$

(iii) $\dfrac{\vdash \Phi_0 \qquad \vdash \Phi_0 \;\Rightarrow\; \Phi_1}{\vdash \Phi_1}$ **(Modus Ponens)**

Similarly the quantifier axioms are all standard [4] except for (**Q.iv**) which asserts that ineffectual operations have no allocation effect, that is they don't change the domain of the state context.
Quantifier Rules (Q).

(i) $\dfrac{\vdash \Phi}{\vdash (\forall x)\Phi}$ **(Generalization $\forall$I)**

(ii) $\vdash (\forall x)(\Phi_0 \;\Rightarrow\; \Phi_1) \;\Rightarrow\; (\Phi_0 \;\Rightarrow\; (\forall x)\Phi_1)$ if $x \notin \mathrm{FV}(\Phi_0)$

(iii) $\vdash (\forall x)\Phi \;\Rightarrow\; \Phi$

(iv) $\vdash (\forall x)\,\mathtt{let}\,\{z := \vartheta(\bar{y})\}\Phi \;\Rightarrow\; \mathtt{let}\,\{z := \vartheta(\bar{y})\}(\forall x)\Phi$

Where in (**Q.iv**) we require that ϑ be ineffectual (see Definition 2.29), $z \neq x$, and $x \notin \mathrm{FV}(\vartheta(\bar{y}))$.

Example 5.4: Note that by the same sort of reasoning that established the derived principle (**Substitution**) we can also establish

(i) $\vdash (\forall x)\,\texttt{let}\,\{z := v\}\Phi \;\Rightarrow\; \texttt{let}\,\{z := v\}(\forall x)\Phi$ for $z \neq x$

5.4. Undefinedness Axiom. The most basic principle concerning undefinedness is that two undefined terms are both equivalued and operationally indistinguishable, (**U.i**).
Undefinedness Axiom (U).

(i) $\vdash {\uparrow} e_0 \;\Rightarrow\; ({\uparrow} e_1 \;\Leftrightarrow\; e_0 \simeq e_1)$

5.5. Constraint Propagation Axioms. An important class of axioms are those which allow assertions to be propagated into and out of assertions. In order to be succinct we write $\Phi[\pm\phi]$ to abbreviate the two formulas $\Phi[\phi]$ and $\Phi[\neg\phi]$. The following three principles require that $\vartheta \in \mathbf{O}$ is ineffectual, Definition 2.29, and that $x, \bar{z}$ distinct.
Constraint Propagation Axioms (S).

(i) $\vdash \texttt{let}\,\{x := \vartheta(\bar{z})\}(x \simeq \vartheta(\bar{z}))$

(ii) $\vdash \pm(z_0 \simeq z_1) \;\Rightarrow\; \texttt{let}\,\{x := \vartheta(\bar{y})\}(\pm(z_0 \simeq z_1))$

(iii) $\vdash \vartheta(\bar{y}){\downarrow} \;\Rightarrow\; (\texttt{let}\,\{x := \vartheta(\bar{y})\}(\pm(z_0 \simeq z_1)) \;\Rightarrow\; \pm(z_0 \simeq z_1))$

5.6. Classes. As a consequence of the semantics of classifications the following are valid.
Class Axioms (Cl).

(i) $\vdash e \in K \;\Rightarrow\; {\downarrow} e$

(ii) $\vdash (\forall\kappa)\Phi[\kappa] \;\Rightarrow\; \Phi[K]$ where Φ contains no `let`-assertions

(iii) $\vdash (\forall x)(x \in \{x \mid \Phi\} \;\Leftrightarrow\; \Phi)$

Example 5.5: As pointed out in the earlier work [22], neither of the following two principles are valid.

(i) $e_0 \cong e_1 \wedge \texttt{let}\,\{x := e_0\}x \in K \;\Rightarrow\; \texttt{let}\,\{x := e_1\}x \in K$

(ii) $(\forall\kappa)\Phi[\kappa] \;\Rightarrow\; \Phi[K]$

For example (i) fails in the case of simple memory operations for class terms can observe inaccessible cells if the expressions differ only in that one generates a cell and forgets it (garbage) and the other does not. (ii) fails because evaluation of class terms (containing `let`-assertions can change the state context in which part of the formula is evaluated.

The axioms and rules above are sound in the sense that a provable formula is valid.

Theorem 5.6 (Soundness)**:** $\vdash \Phi$ implies that $\models \Phi$.

§**6.** Λ_{s} **– An Example.** As a particular example of a Landinesque language we define Λ_{s}, a language extending the call-by-value lambda calculus with the usual functional programming primitives, primitives for the manipulation of memory, and a control primitive for manipulating the current continuation (the reduction context). This combination results in a language similar to "functional" programming languages such as Scheme [38] and ML [32]. The combination of lambda abstraction, basic data structures such as numbers and pairing, memory and control primitives provides a basis for expressing a wide range of programming paradigms/styles quite naturally. Examples of the use of continuations in programming practice can be found in [14, 16].

In Λ_{s} the set of atoms, $\mathbf{A}$, contains two distinct atoms playing the role of booleans, `t` for *true* and `nil` for *false*, as well as atoms playing the role of the natural numbers, which we denote by $0, 1, \ldots$.

Definition 6.1 (Λ_{s} Operations)**:**

$$\mathbf{O} = \mathbf{O}^{\mathrm{f}} \cup \mathbf{O}^{\mathrm{m}} \cup \mathbf{O}^{\mathrm{c}}$$

$$\mathbf{O}^{\mathrm{f}} = \{\mathtt{app}, \mathtt{lambda?}, \mathtt{br}, \mathtt{eq?}, \mathtt{atom?}, \mathtt{pr}, \mathtt{fst}, \mathtt{snd}, \mathtt{pr?}, +\mathtt{1}, -\mathtt{1}, \mathtt{nat?}\}$$

$$\mathbf{O}^{\mathrm{m}} = \{\mathtt{mk}, \mathtt{get}, \mathtt{set}, \mathtt{cell?}\}$$

$$\mathbf{O}^{\mathrm{c}} = \{\mathtt{ncc}\}$$

6.1. Informal Semantics. We give a brief and informal guide to the more novel of the primitive operations.

`mk` is a memory allocation primitive: the evaluation of $\mathtt{mk}(v)$ results in the allocation of a *new memory cell* and initializes this cell so that it contains the value v. The value returned by this call to `mk` is the newly allocated cell. `mk` is total, it has effects.

`get` is the memory access primitive: the evaluation of $\mathtt{get}(v)$ is defined iff v is a memory cell. If v is a memory cell, then $\mathtt{get}(v)$ returns the value stored in that cell. Note that there is no reason why a cell cannot store itself (or some more elaborate cycle). `get` is both partial and ineffectual.

`set` is the memory modification primitive: the evaluation of $\mathtt{set}(v_0, v_1)$ is defined iff v_0 is a memory cell. If v_0 is a memory cell, then $\mathtt{set}(v_0, v_1)$ modifies that cell so that its new contents becomes v_1. The value returned by a call to `set` is somewhat arbitrary and somewhat irrelevant. We have chosen `nil` as the return value, thus if v is a cell, then $\mathtt{set}(v, v)$ will return `nil`, and more importantly modify v so that it contains itself. `set` is both partial and has effects.

`cell?` is the recognizer of the set of memory cells. $\mathtt{cell?}(v)$ returns `t` if v is a memory cell, otherwise it returns `nil`. It is both total and ineffectual.

`br` is the strict branching primitive: the evaluation of $\mathtt{br}(v_0, v_1, v_2)$ returns v_2 if v_0 is the atom `nil`, otherwise it returns v_1. Thus any non-`nil` value is considered *true*. The usual lazy branching primitive

$$\mathtt{if}(e_0, e_1, e_2) \stackrel{\triangle}{=} \mathtt{app}(\mathtt{br}(e_0, \lambda z.e_1, \lambda z.e_2), \mathtt{nil}$$

for a fresh variable z. br is total and ineffectual.

eq? is the equality primitive (solely on atoms): $\mathtt{eq?}(v_0, v_1)$ returns t if v_0 and v_1 are the same atom, otherwise it returns nil. eq? is total and ineffectual.

ncc is a primitive for capturing the current continuation (i.e the current reduction context). It gets its name from what it does, since it **n**otes the **c**urrent **c**ontinuation. $\mathtt{ncc}(v)$ captures the current reduction context as a continuation, removes this reduction context from the current state, and applies v to the captured continuation at the top level. ncc, like app, is both partial and has effects.

Definition 6.2 (More notation conventions)**:** We introduce new constant symbols by definitional extension using the notation

$$\mathtt{c} \stackrel{\triangle}{=} v$$

We also use this notation for introducing function symbols defined recursively. Thus $\mathtt{f} \stackrel{\triangle}{=} \lambda x.e$ stands for the following definition of the constant symbol f

$$\mathtt{f} \stackrel{\triangle}{=} \mathtt{Y}(\lambda \mathtt{f}.\lambda x.e)$$

where Y is some choice of call-by-value Y-combinator, see for example § 6.2.2.

We use a number of abbreviations to (hopefully) make programs more readable.

(top): ncc can be used to define an abort primitive. We call this top because it simply returns its argument to the top level.

$$\mathtt{top} \stackrel{\triangle}{=} \lambda x.\mathtt{ncc}(\lambda k.x)$$

(Sequencing): We let $e_0; e_1$ abbreviate $\mathtt{let}\,\{d := e_0\}e_1$ for some d not free in e_1. This expresses the sequencing of two expressions, the first being evaluated for effect only – for example to update memory. By the Λ_c laws (see Example 2.28), we see that ; so defined is associative, thus we may write $e_0; \ldots ; e_n$ without ambiguity.

(app suppression): As in the usual λ-calculus notation, we replace app by juxtaposition and well-placed parentheses, and when convenient, curry application of a function to multiple arguments. Thus $e_0(e_1)$ abbreviates $\mathtt{app}(e_0, e_1)$ and $e_0(e_1, e_2)$ abbreviates $\mathtt{app}(\mathtt{app}(e_0, e_1), e_2)$.

Definition 6.3 (Λ_s States)**:** In Λ_s states are memory contexts—contexts of the form

$$\begin{aligned}\zeta = {} & \mathtt{let}\,\{z_1 := \mathtt{mk}(\mathtt{nil})\} \\ & \quad \ldots \\ & \quad \mathtt{let}\,\{z_k := \mathtt{mk}(\mathtt{nil})\} \\ & \qquad \mathtt{set}(z_1, v_1); \ldots \mathtt{set}(z_k, v_k); \circ^{\{z_1 \mapsto z_1, \ldots, z_k \mapsto z_k\}}\end{aligned}$$

The lets of ζ allocate new cells, named z_i, and the sets assign the contents. If the value put in a cell does not refer to any newly created cell then that set could be omitted and the value expression used as argument to the corresponding mk. However in general, separation of allocation and assignment is needed in order to represent stores with cycles. For example, consider creating a cell that contains itself. This is described by the memory context $\mathtt{let}\,\{z := \mathtt{mk}(\mathtt{nil})\}\mathtt{set}(z, z); \circ^{\{z \mapsto z\}}$. This is not the same as the context $\mathtt{let}\{z := \mathtt{mk}(z)\}\circ^{\{z \mapsto z\}}$, since in the latter case the z in the argument to mk is bound outside the context and is necessarily distinct from the created z.

Note that for ζ in the above form, $\mathrm{Dom}(\zeta) = \{z_1, \ldots z_k\}$ as defined in Definition 2.6. Memory contexts are a syntactic representation of the stores of more traditional semantics (finite maps from locations to storable values). Thus, we define analogs of finite map operations on memory contexts. $\zeta(z_i) = v_i$ for $1 \le i \le k$, and we write $\{z_i := \mathtt{mk}(v_i) \bigm| 1 \le i \le k\}$ for ζ. We also write $\{z_i := \mathtt{mk}(v_i) \bigm| 1 \le i \le k\}e$ for $\zeta^{\circ \mapsto e}$. $\zeta\{z := \mathtt{mk}(v)\}$ is the memory context, ζ' with domain $\mathrm{Dom}(\zeta) \cup \{z\}$, such that $\zeta'(z) = v$ and $\zeta'(z') = \zeta(z')$ for $z' \in \mathrm{Dom}(\zeta) - \{z\}$. This notation is intentionally ambiguous about the order of allocation of cells and assigning values to cells. When we only care about the finite map represented by ζ the ambiguity makes no difference.

Definition 6.4 (Λ_{s} Reduction Rules)**:** The more interesting Λ_{s} reduction rules include the β_{v} rule given in Definition 2.18 and the following reduction rules for the memory and control operations:

$$\begin{array}{lll}
(\mathrm{mk}) & \zeta : R[\mathtt{mk}(v)] \longrightarrow \zeta\{z := \mathtt{mk}(v)\} : R[z] & \text{for } z \text{ fresh} \\
(\mathrm{get}) & \zeta\{z := \mathtt{mk}(v)\} : R[\mathtt{get}(z)] \longrightarrow \zeta\{z := \mathtt{mk}(v)\} : R[v] & \\
(\mathrm{set}) & \zeta\{z := \mathtt{mk}(v')\} : R[\mathtt{set}(z, v)] \longrightarrow \zeta\{z := \mathtt{mk}(v)\} : R[\mathtt{nil}] & \\
(\mathrm{ncc}) & \zeta : R[\mathtt{ncc}(v)] \longrightarrow \zeta : \mathtt{app}(v, \lambda x.\mathtt{top}(R[x])) & \text{for } x \notin \mathrm{FV}(R).
\end{array}$$

6.2. Some Example Programs and Computations. We begin with two simple calculations, just to illustrate the basics of cell and continuation manipulation.

6.2.1. *Warmup exercises.*

Example 6.5 (Memory Operations)**:** Evaluation of the expression

$$\mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\mathtt{set}(z, 0); \mathtt{get}(z)$$

creates a new cell named z, stores 0 in that cell, then retrieves the value. The calculation goes as follows:

$\zeta : \mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\mathtt{set}(z, 0); \mathtt{get}(z)$

$\longrightarrow \zeta\{z := \mathtt{mk}(\mathtt{nil})\} : \mathtt{let}\,\{z := z\}\mathtt{set}(z, 0); \mathtt{get}(z)$

by applying the mk rule with R being $\mathtt{let}\,\{z := \bullet\}\mathtt{set}(z, 0); \mathtt{get}(z)$, and assuming without loss of generality that $z \notin \mathrm{Dom}(\zeta)$

$\longrightarrow \zeta\{z := \mathtt{mk}(\mathtt{nil})\} : \mathtt{set}(z, 0); \mathtt{get}(z)$

by the `let` rule, i.e. β_{v}, and the definition of `let`

$\longrightarrow \zeta\{z := \mathtt{mk}(0)\} : \mathtt{nil}; \mathtt{get}(z)$

by the `set` rule

$\longrightarrow \zeta\{z := \mathtt{mk}(0)\} : \mathtt{get}(z)$

by the `seq` rule, i.e. the `let` rule using the definition of ;.

$\longrightarrow \zeta\{z := \mathtt{mk}(0)\} : 0$

by the `get` rule

Example 6.6 (`top`)**:** Next we justify the claim that `top` is an abort primitive by showing that

$$\zeta : R[\![\mathtt{top}(v)]\!] \longrightarrow\!\!\!\!\rightarrow\!\!\succ \zeta : v.$$

This is a simple calculation using the computation rules

$\zeta : R[\![\mathtt{top}(v)]\!] \overset{\triangle}{=} \zeta : R[\![(\lambda x.\mathtt{ncc}(\lambda k.x))(v)]\!]$ using the definition of `top`

$\longrightarrow \zeta : R[\![\mathtt{ncc}(\lambda k.v)]\!]$ by β_{v}, $k \notin \mathrm{FV}(v)$

$\longrightarrow \zeta : (\lambda k.v)(\lambda x.\mathtt{top}(R[\![x]\!]))$ by the `ncc` rule

$\longrightarrow \zeta : v$ again by β_{v}

6.2.2. *Landin's Recursion Operator.* Since the Λ_{s}-language extends the call-by-value λ-calculus the usual call-by-value fixed point combinator is a term in the language. A somewhat different fixed point combinator, that makes use of the reference primitives, is possible:

$$\begin{aligned}\mathtt{Y} \overset{\triangle}{=} \lambda y.\, &\mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\} \\ &\quad \mathtt{set}(z, \lambda x.\mathtt{app}(\mathtt{app}(y, \mathtt{get}(z)), x)); \mathtt{get}(z))\end{aligned}$$

This version of the fixed-point combinator is similar in spirit to the one suggested by Landin [25] and is operationally equivalent to the usual call-by-value Y combinator [29]. When applied to a functional F of the form $\lambda f.\lambda x.e$, `Y` creates a private local cell, z, with contents $G = \lambda x.\mathtt{app}(\mathtt{app}(F, \mathtt{get}(z)), x)$, and returns G. By privacy of z, G is equivalent to $F(G)$ (cf. [29]). Note that this example is typable in the simply typed lambda calculus (for provably non-empty types (cf. [22])). Thus adding operations for manipulating references to the simply typed lambda calculus causes the failure of strong normalization as well as many other of its nice mathematical properties.

6.2.3. *Integer Streams.* From an abstract point of view, a stream is simply a (possibly infinite) sequence of data [1]. In the Λ_{s}-language we can represent streams simply as functional objects, lambda expressions with free variables bound to cells. The sequence corresponding to a Λ_s-stream is the values returned by repeated application of the object to a fixed (and hopefully irrelevant) argument. The simplest example of a non-trivial Λ_s-stream is the stream of natural numbers.

```
makeStream ≜ λm. let {z := mk(m)}
                 λx. let{n := get(z)}
                       set(z, n + 1); n
```

Here `makeStream` applied to an integer m creates a stream of integers beginning with that integer. The so-created stream when queried (applied to any value) returns the next integer in the stream.

6.2.4. *The Sieve of Eratosthenes.* A somewhat more interesting example of a stream is the sieve of Eratosthenes [1]. We begin with the functional `filter` which expects an integer, n, and a stream, s and then creates a new stream. This new stream when queried repeatedly calls the stream argument, s, until an integer not divisible by the number argument, n, is returned. This number is then returned as the answer to the query.

```
filter ≜ λn.λs.
             λx. let{m := s(nil)}
                   if(divides?(n, m)
                      filter(n, s)(nil)
                      m)
```

`sieve` is an expression which when evaluated creates a new sieve of Eratosthenes. This new stream is a stream of the prime numbers. Each time the stream is queried it returns the current prime and updates its local stream to filter with this prime.

```
sieve ≜ let{sc := mk(makeStream(2))}
             λx. let{s := get(sc)} let {p := s(nil)}
                   set(sc, filter(p, s)); p
```

6.2.5. *Co-routines.* Co-routines are a programming paradigm useful for incremental processing of streams of data. Typical examples of their use come from compiler construction, and more recently from managing data coming in segments over a network connection. Each time an additional increment of data is needed/available, the co-routine is *resumed*. The co-routine computes the next increment, remembers where it left off, and returns the incremental result. From a programming language point of view what is needed is language constructs that make it easy to remember the current state, and to continue processing when next invoked. The high-level structure of a co-routine program might look like

```
          initialize
    loop: get the next big block of data;
          process segment 1 giving result-1;
```

```
            resume with result-1;
              ...
            process segment n giving result-n;
            resume with result-n;
            repeat loop;
```

where `resume` implicitly saves the point in the program where the computation suspended for later resumption and returns control to the resumer of the co-routine. Thus from the point of view of the co-routine, the partner also appears to be a co-routine.

A resumption primitive can be defined using our control and memory primitives as follows.

$$\mathtt{resume} \overset{\triangle}{=} \lambda z.\lambda v.\mathtt{ncc}(\lambda k.\,\mathtt{let}\,\{c := \mathtt{get}(z)\}\mathtt{set}(z,k); c(v))$$

It is assumed that z is a cell where the resumee's resumption point is stored (as a continuation). $\mathtt{resume}(z)(v)$ gets that state, temporarily calling it c, saves the resumer's resumption point, captured by $\mathtt{ncc}$ and bound to k, in z, and starts at its resumption point with argument v. The resumed program should eventually call $\mathtt{resume}(z)(w)$ where w is the next increment to be returned to the resumer. Later we will show using a simple example how the Feferman–Landin logic allows us to reason about co-routines. For now we look at a sample calculation and an example co-routine. Assume that q is the resumption point of a partner co-routine that is being resumed in a context R, then

$$\begin{aligned}
&\zeta\{z := \mathtt{mk}(q)\} : R[\![\mathtt{resume}(z)(v)]\!]\\
&\quad \longrightarrow\!\!\!\succ \zeta\{z := \mathtt{mk}(q)\} : R[\![\mathtt{ncc}(\lambda k.\,\mathtt{let}\,\{c := \mathtt{get}(z)\}\mathtt{set}(z,k); c(v))]\!]\\
&\quad \longrightarrow\!\!\!\succ \zeta\{z := \mathtt{mk}(q)\} : \mathtt{let}\,\{c := \mathtt{get}(z)\}\mathtt{set}(z, \lambda x.\mathtt{top}(R[\![x]\!])); c(v))\\
&\quad \longrightarrow\!\!\!\succ \zeta\{z := \mathtt{mk}(q)\} : \mathtt{set}(z, \lambda x.\mathtt{top}(R[\![x]\!])); q(v)\\
&\quad \longrightarrow\!\!\!\succ \zeta\{z := \mathtt{mk}(\lambda x.\mathtt{top}(R[\![x]\!]))\} : q(v)
\end{aligned}$$

Thus we have the derived computation rule:

$$\zeta\{z := \mathtt{mk}(q)\} : R[\![\mathtt{resume}(z)(v)]\!] \longrightarrow\!\!\!\succ \zeta\{z := \mathtt{mk}(\lambda x.\mathtt{top}(R[\![x]\!]))\} : q(v)$$

now suppose

$$\begin{aligned}
&\zeta\{z := \mathtt{mk}(\lambda x.\mathtt{top}(R[\![x]\!]))\} : q(v)\\
&\quad \longrightarrow\!\!\!\succ \zeta'\{z := \mathtt{mk}(\lambda x.\mathtt{top}(R[\![x]\!]))\} : R'[\![\mathtt{resume}(z)(w)]\!]
\end{aligned}$$

then applying the above derived reduction we have

$$\zeta\{z := \mathtt{mk}(q)\} : R[\![\mathtt{resume}(z)(v)]\!] \longrightarrow\!\!\!\succ \zeta'\{z := \mathtt{mk}(\lambda x.\mathtt{top}(R'[\![x]\!]))\} : R[\![w]\!]$$

with the co-routine's next resumption point saved in the cell z.

Here is a simple co-routine (schema):

```
$\mathtt{coex} \overset{\triangle}{=} \lambda i.\lambda z.\lambda x.$
          $\mathtt{let}\,\{p := \mathtt{next}(i)\}$
            $\mathtt{let}\,\{x_1 := f_1(p)\}$
              $\mathtt{resume}(z)(x_1);$
              $\mathtt{let}\,\{x_2 := f_2(p)\}$
                $\mathtt{resume}(z)(x_2);$
                $\mathtt{coex}(i,z)(\mathtt{nil})$
```

The initial resumption point for the co-routine is of the form $\lambda x.\mathtt{top}(\mathtt{coex}(i,z)(x))$, where i is intended to be a cell containing access to an input stream, which we leave unspecified, except that $\mathtt{next}(i)$ must produce data in the domain of f_1 and f_2, which compute functions without any use of or effect on state.

Now consider a program that does the same incremental computation but explicitly saves resumption point information.

```
$\mathtt{stex} \overset{\triangle}{=} \lambda i \lambda z.\lambda x.$
          $\mathtt{let}\,\{s := \mathtt{get}(z)\}$
            $\mathtt{if}(\mathtt{eq?}(\mathtt{fst}(s),0),$
               $\mathtt{let}\,\{p := \mathtt{next}(i)\}\mathtt{set}(z,\mathtt{pr}(1,p)); f_1(p),$
               $\mathtt{let}\,\{p := \mathtt{snd}(s)\}\mathtt{set}(z,\mathtt{pr}(0,\mathtt{nil})); f_2(p))$
```

Then we have the following lemma. We will see later how to establish such claims in the Feferman–Landin logic.

Lemma 6.7:

$$\{z := \mathtt{mk}(\lambda x.\mathtt{top}(\mathtt{coex}(i,z)(x)))\}\mathtt{resume}(z) \cong \{z := \mathtt{mk}(\mathtt{pr}(0,\mathtt{nil}))\}\mathtt{stex}(i,z)$$

6.3. Axioms for Λ_s. The first, most basic axiom concerning operational equivalence and equivaluedness for Λ_s, is that the booleans $\mathtt{t}$ and $\mathtt{nil}$ are not equivalent.
Non-Triviality (T).

(i) $\vdash \neg(\mathtt{t} \simeq \mathtt{nil})$

To the operational equivalence axioms we can add a *garbage collection* principle, and a principle expressing the simple fact that operational equivalence and equivaluedness coincide on simple data.
Operational Equivalence Axioms ($\cong$).

(iii) $\vdash e \cong \zeta\,[\![e]\!]$ provided $\mathrm{FV}(e) \cap \mathrm{Dom}(\zeta) = \emptyset$

(iv) $\vdash \tau(x) \sim \mathtt{t} \wedge \tau(y) \sim \mathtt{t} \;\Rightarrow\; (x \cong y \Leftrightarrow x \sim y)$ $\tau \in \{\mathtt{atom?}, \mathtt{cell?}, \mathtt{nat?}\}$

Next we extend the quantifier and constraint propagation axioms of § 5 using the information we have about particular Λ_s operations.

Quantifier Axioms (Q).

(iv) $\vdash (\forall x)\, \mathtt{let}\, \{z := \vartheta(\bar{y})\}\Phi \;\Rightarrow\; \mathtt{let}\, \{z := \vartheta(\bar{y})\}(\forall x)\Phi$

ϑ is either ineffectual or the operation `set`

(v) $\vdash ((\forall x)\, \mathtt{let}\, \{z := \mathtt{mk}(y)\}\Phi(x) \;\wedge\; \mathtt{let}\, \{z := \mathtt{mk}(y)\}\Phi(z))$
$\quad\Rightarrow\; \mathtt{let}\, \{z := \mathtt{mk}(y)\}(\forall x)\Phi$

Constraint Propagation Axioms (S). The following two principles require that $\vartheta_0, \vartheta_1 \in \mathbf{O}$ and that if $\vartheta_1 \in \{\mathtt{set}, \mathtt{app}, \mathtt{ncc}\}$, then $\vartheta_0 \in \mathbf{O} - \{\mathtt{get}, \mathtt{set}, \mathtt{app}, \mathtt{ncc}\}$.

(iv) $\vdash \pm(z \simeq \vartheta_0(\bar{y})) \;\Rightarrow\; \mathtt{let}\, \{x := \vartheta_1(\bar{w})\}(\pm(z \simeq \vartheta_0(\bar{y})))$

(v) $\vdash \vartheta_1(\bar{w})\!\downarrow \;\Rightarrow\; (\mathtt{let}\, \{x := \vartheta_1(\bar{w})\}(\pm(z \simeq \vartheta_0(\bar{y}))) \;\Rightarrow\; \pm(z \simeq \vartheta_0(\bar{y})))$

Note that in (**S.iv,v**) `ncc` has the same classification as `app`, since as noted above it applies its argument lambda to a value and thus can reduce to an arbitrary expression eventually producing arbitrary effects. Also note that taking ϑ_1 to be `mk`, (**S.iv**) says that allocation, does not change properties of any operations on existing values.

6.4. Definedness Principles. We have already stated the most basic principle concerning undefinedness, namely that two undefined terms are both equivalued and operationally indistinguishable, (**U.i**). The rest of the principles concern the partiality of the underlying operations. Note that in the case of the memory operations `mk` and `set`, being defined is not the same as being equivalent to a value. In the other cases this is true, although we need only express the weaker form. The stronger forms are derivable.

Definedness rules (D).

(i) $\vdash \downarrow\!\mathtt{mk}(z)$

(ii) $\vdash \downarrow\!\mathtt{set}(z, x) \;\Leftrightarrow\; \mathtt{cell?}(z) \simeq \mathtt{t}$

(iii) $\vdash \downarrow\!\mathtt{get}(x) \;\Leftrightarrow\; \mathtt{cell?}(x) \simeq \mathtt{t} \;\Leftrightarrow\; (\exists y)(\mathtt{get}(x) \simeq y)$

(iv) $\vdash \downarrow\!\mathtt{ncc}(x) \;\Rightarrow\; \mathtt{lambda?}(x) \simeq \mathtt{t}$

(v) $\vdash \downarrow\!\mathtt{app}(x, y) \;\Rightarrow\; \mathtt{lambda?}(x) \simeq \mathtt{t}$

(vi) $\vdash \downarrow\!\vartheta(\bar{x})$

Where in (**D.vi**) we assume that $\vartheta \in \{\mathtt{atom?}, \mathtt{cell?}, \mathtt{lambda?}, \mathtt{nat?}, \mathtt{pr?}, \mathtt{pr}, \mathtt{eq?}, \mathtt{br}\}$ and $\bar{x}$ is the appropriate length.

6.5. Axioms for Memory. The principles concerning `mk` are quite straightforward. (**mk.i**) describes the allocation effect of a call to `mk`. The remaining two principles (**mk.ii**) and (**mk.iii**) assert that the time of allocation has no discernable effect on the resulting call, however since we are in a world with control effects e_0 must be free of them for these two principles to be valid [12].

Allocation Axioms (mk).

(i) $\vdash \mathtt{let}\, \{x := \mathtt{mk}(z)\}(\neg(x \cong y) \;\wedge\; \mathtt{cell?}(x) \cong \mathtt{t} \;\wedge\; \mathtt{get}(x) \cong z)$

(ii) $\vdash \mathtt{let}\,\{y := e_0\}\,\mathtt{let}\,\{x := \mathtt{mk}(z)\}e_1 \cong \mathtt{let}\,\{x := \mathtt{mk}(z)\}\,\mathtt{let}\,\{y := e_0\}e_1$

(iii) $\vdash \mathtt{let}\,\{y := e_0\}\,\mathtt{let}\,\{x := \mathtt{mk}(z)\}\Phi \Leftrightarrow \mathtt{let}\,\{x := \mathtt{mk}(z)\}\,\mathtt{let}\,\{y := e_0\}\Phi$

The principle (**mk.i**) requires that $x \not\in \{z, y\}$, while both (**mk.ii**) and (**mk.iii**) require that e_0 is closed with no control operations.

The first two contextual assertions regarding `set` are analogous to those of (`mk`**.i**). They describe what is returned and what is altered. The remaining four principles involve the commuting, cancellation, absorption, and idempotence of calls to `set`. For example the `set` absorption principle, (`set`**.v**), expresses that under certain simple conditions allocation followed by assignment may be replaced by a suitably altered allocation.

Modification Axioms (set).

(i) $\vdash \mathtt{cell?}(z) \Rightarrow \mathtt{let}\,\{x := \mathtt{set}(z, y)\}(\mathtt{get}(z) \cong y \wedge x \cong \mathtt{nil})$

(ii) $\vdash \mathtt{get}(x) \cong y \Rightarrow \mathtt{set}(x, y) \cong \mathtt{nil}$

(iii) $\vdash \neg(x_0 \cong x_2) \Rightarrow \mathtt{set}(x_0, x_1); \mathtt{set}(x_2, x_3) \cong \mathtt{set}(x_2, x_3); \mathtt{set}(x_0, x_1)$

(iv) $\vdash \mathtt{set}(x, y_0); \mathtt{set}(x, y_1) \cong \mathtt{set}(x, y_1)$

(v) $\vdash \mathtt{let}\,\{z := \mathtt{mk}(x)\}\mathtt{set}(z, w); e \cong \mathtt{let}\{z := \mathtt{mk}(w)\}e$ if $z \not\in \mathrm{FV}(w)$

(vi) $\vdash \mathtt{get}(x) \cong y \Rightarrow \mathtt{set}(x, y) \cong \mathtt{nil}$

Accessor Axioms (get).

(i) $\vdash \mathtt{let}\,\{x := \mathtt{get}(y)\}(x \cong \mathtt{get}(y))$

(ii) $\vdash \Phi \Rightarrow \mathtt{let}\,\{x := \mathtt{get}(y)\}\Phi$ $x \not\in \mathrm{FV}(\Phi)$

6.6. Axioms for Control. In our model a continuation is a function that returns a value to the top level, thus continuations can be modeled by composition of `top` with an arbitrary lambda expression. The first control axiom says that within the immediate context of `ncc`, the current continuation is `top`, while the second shows how explict R and implicit (passed to the argument of `ncc`) continuations are related, and the third axiom expresses the fact that the argument of `ncc` is always applied to a continuation. The fourth axiom says that $\mathtt{ncc}(\lambda c.R[\bullet])$ behaves like a reduction context, in the sense that any expression occupying the hole is alway evaluated next, and thus can be pulled out using `let` analogous to (**let.ii**) as long as the continuation variable does not appear free. The fifth axiom expresses the fact that capturing the current continuation and then re-installing it is equivalent the identity operation.

Control Axioms (ncc).

(i) $\vdash \mathtt{ncc}(\lambda c.\mathtt{ncc}(e)) \cong \mathtt{ncc}(\lambda c.\mathtt{app}(e, \mathtt{top}))$

(ii) $\vdash R[\mathtt{ncc}(e)] \cong \mathtt{ncc}(\lambda c.\mathtt{app}(e, \lambda x.c(R[x])))$

if $c \not\in \mathrm{FV}(e, R)$ and $x \not\in \mathrm{FV}(c, R)$

(iii) $\vdash \mathtt{ncc}(\lambda c.C[c]) \cong \mathtt{ncc}(\lambda c.C[\lambda x.\mathtt{top}(c(x))])$

provided $c \notin \text{Traps}(C)$.

(iv) $\vdash (\texttt{let}\,\{x := e\}\texttt{ncc}(\lambda k.R[\![x]\!])) \cong \texttt{ncc}(\lambda k.R[\![e]\!])$

if x is fresh, and $k \notin \text{FV}(e)$

(v) $\vdash \texttt{ncc}(\lambda k.k(e)) \cong e \qquad \text{if } k \notin \text{FV}(e)$

To illustrate contextual reasoning we derive a useful law for reasoning about `ncc`.

Lemma 6.8 (ncc.vi)**:**

(vi.a) $\vdash e_0 \cong e_1 \;\Rightarrow\; (\texttt{ncc}(\lambda k.R[\![e_0]\!]) \cong \texttt{ncc}(\lambda k.R[\![e_1]\!]))$

$k \notin \text{FV}(e_i)$ for $i < 2$.

(vi) $\vdash \texttt{let}\,\{x := e\}(e_0 \cong e_1) \;\Rightarrow\; \texttt{let}\,\{x := e\}(\texttt{ncc}(\lambda k.R[\![e_0]\!]) \cong \texttt{ncc}(\lambda k.R[\![e_1]\!]))$

if $k \notin \text{FV}(e)$

Proof: To show (**ncc.vi.a**) we argue as follows.

$e_0 \cong e_1 \;\Rightarrow\; \texttt{let}\,\{x := e_0\}\texttt{ncc}(\lambda k.R[\![x]\!]) \cong \texttt{let}\,\{x := e_1\}\texttt{ncc}(\lambda k.R[\![x]\!])$

by (**E.iv**)

$\texttt{ncc}(\lambda k.R[\![e_j]\!]) \cong \texttt{let}\,\{x := e_j\}\texttt{ncc}(\lambda k.R[\![x]\!])$

for $j < 2$, by (**ncc.iv**)

And by a little propositional reasoning we are done.

(**ncc.vi**) follows from (**ncc.vi.a**) using context introduction (**C.i**). □

Lemma 6.9 (top)**:** Any reduction context surrounding an application of `top` can be discarded.

$$R[\![\texttt{top}(e)]\!] \cong \texttt{top}(e)$$

Proof: This follows from (ncc.iv) using the `let` laws. □

6.7. Simulation Principle for Λ_s. Now we introduce a simulation principle that allows us to prove equivalence of two lambdas with memory. The principle says that to show such an equivalence it is sufficient to find a correspondence between memories such that whenever started in corresponding states and applied to the same value, both lambdas

(i) return the same (first-order) value
(ii) have corresponding effects on their memory states

To simplify notation, we present only a very special case of the principle, which is adequate to illustrate the basic idea. First we must define the set of first-order values and a function that maps any value to its first-order part.

Definition 6.10 (First-orderness, (**FO**, `fo`))**:** **FO** is the class of first-order values. This is the least class containing **At** and closed under pairing (essentially the well-founded

S-expressions of [22]). It can be defined using the standard means of giving inductive definitions in variable typed theories as follows.

$$\mathbf{FO} \overset{\triangle}{=} \bigcap_{\kappa}(T_{\mathbf{FO}}[\kappa] \subseteq \kappa)$$

$$\text{where} \quad T_{\mathbf{FO}}[\kappa] \overset{\triangle}{=} \{x \mid \mathtt{atom?}(x) \lor (\mathtt{pr?}(x) \land \mathtt{fst}(x) \in \kappa \land \mathtt{snd}(x) \in \kappa)\}$$

$$\text{and} \quad \bigcap_{\kappa} \Phi[\kappa] \overset{\triangle}{=} \{x \mid (\forall \kappa)(\Phi[\kappa] \Rightarrow x \in \kappa)\}$$

The operation, `fo`, that strips away the non first order parts of its argument. It leaves atoms unchanged, commutes with pairing, and maps everything else (lambdas and cells) to `nil`.

$$\mathtt{fo} \overset{\triangle}{=} \lambda x.\mathtt{if}(\mathtt{atom?}(x), x, \mathtt{if}(\mathtt{pr?}(x), \mathtt{pr}(\mathtt{fo}(\mathtt{fst}(x)), \mathtt{fo}(\mathtt{snd}(x))), \mathtt{nil}))$$

Using the induction principle derived from the inductive definition of **FO** it is easy to show $\mathbf{FO} \subseteq \{x \mid \mathtt{fo}(x) \simeq x\}$. To prove the converse we need the least-fixed-point principle derived from recursive definitions using a suitable fixed-point combinator.

We state the simulation principle for the special case in which the lambda's local memory is a single cell. In the definition below we use z to name this local memory cell. Since we want the cell contents to be able to refer to the cell, the correspondence Ψ is formulated as a relation on **L** such that corresponding lambdas are applied to a cell to produce the actual corresponding values ($y_j(z)$). Similarly, the lambda's P_j represent abstractions over the cell variable. The assertions (**Sim.1**) and (**Sim.2**) are formal statements of the requirements (i) and (ii) given at the beginning of this subsection.

Simulation Principle (SP). If z a variable, $\Psi \subseteq \mathbf{L} \times \mathbf{L}$, and P_j is a lambda expression for $j < 2$, such that z is not free in Ψ, P_0 or P_1 and

$$(\forall (y_0, y_1) \in \Psi)(\forall x)(\exists (y'_0, y'_1) \in \Psi, v \in \mathbf{FO})$$

$$\bigwedge_{j \in \{0,1\}}$$

$$(\{z := \mathtt{mk}(y_j(z))\}(P_j(z)(x) \cong \mathtt{set}(z, y'_j(z)); v) \quad \text{(Sim.1)}$$

$$\land$$

$$\{z := \mathtt{mk}(y_j(z))\}(P_j(z)(x) \cong P_j(z)(\mathtt{fo}(x)))), \quad \text{(Sim.2)}$$

then $(\forall (y_0, y_1) \in \Psi)\{z := \mathtt{mk}(y_0(z))\}P_0(z) \cong \{z := \mathtt{mk}(y_1(z))\}P_1(z)$.

6.8. Using the Simulation Principle. Now we show how to establish equivalences such as the claim made in the co-routine lemma(6.7). We simplify the example a bit in order to be able to fill in some details in a reasonable space.

Lemma 6.11 (Simple co-routine reduction to state.)**:** Let

$$\mathtt{inc}(f, n) \overset{\triangle}{=} \lambda z.\lambda x.\mathtt{resume}(z)(f(n)); \mathtt{inc}(f, n+1)(z)(\mathtt{nil})$$

$\mathtt{ino}(f) \overset{\triangle}{=} \lambda z.\lambda x.\,\mathtt{let}\,\{n := \mathtt{get}(z)\}\mathtt{set}(z, n+1); f(n).$

`inc` (c for co-routine) produces a sequence of values $f(n)$ for $n \in \mathbf{N}$ using resumption to remember the next element of the stream to generate. `ino` (o for object) produces the same sequence by explicitly keeping track of the next integer argument in a memory cell. Now choose $f \in \mathbf{FO} \to \mathbf{FO}$ such that $f(x) \cong f(\mathtt{fo}(x))$. Define κ_n to be the lambda term $\lambda x.\mathtt{top}(\mathtt{inc}(f, n)(z)(\mathtt{nil}))$. Then

$$\{z := \mathtt{mk}(\kappa_n)\}\mathtt{resume}(z) \cong \{z := \mathtt{mk}(n)\}\mathtt{ino}(f)(z)$$

Proof: The proof is by the simulation principle with

$$\Psi = \{(\lambda z.\kappa_n, \lambda z.n) \;\big|\; n \in \mathbf{N}\}$$

$$P_0 = \mathtt{resume}$$

$$P_1 = \mathtt{ino}(f)$$

The simulation requirement (**Sim.2**) follows from the assumption on f. Note that by definition 4.5 such f are total and ineffectual. Thus we only need to establish (**Sim.1**) for the two Ps. The argument for $\mathtt{ino}(f)$ is as follows. In the following we write $\zeta \vdash e_0 \cong e_1$ in place of $\vdash \zeta(e_0 \cong e_1)$. This is done to factor out the background contextual reasoning from the basic equational reasoning.

$\{z := \mathtt{mk}(n)\} \vdash$
 $\mathtt{ino}(f)(z)(x)$
  $\cong \mathtt{let}\,\{n := \mathtt{get}(z)\}\mathtt{set}(z, n+1); f(n)$
   by (**let.i**)
  $\cong \mathtt{set}(z, n+1); f(n)$
   by (**mk.i**) and (**let.i**)

The argument for `resume` is a bit more complicated. First we expand and transform $\mathtt{resume}(z)(x)$ in the given memory context.

$\{z := \mathtt{mk}(\kappa_n)\} \vdash$
 $\mathtt{resume}(z)(x)$
  $\cong \mathtt{ncc}(\lambda k.\,\mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z, k); p(x))$
   by the definition of `resume`
  $\cong \mathtt{ncc}(\ \lambda k.\,\mathtt{let}\,\{p := \kappa_n\}\mathtt{set}(z, k); p(x)\ )$
   by (**ncc.vi**) and `get` laws

Now by purely equational reasoning we show that the contextually transformed expression is equivalent to an expression with directly nested `ncc`s. First we simplify the `let` and unfold definitions.

$$
\begin{aligned}
&\mathtt{ncc}(\ \lambda k.\,\mathtt{let}\,\{p := \kappa_n\}\mathtt{set}(z,k); p(x)\ )\\
&\quad\cong \mathtt{ncc}(\ \lambda k.\mathtt{set}(z,k); \mathtt{top}(\mathtt{inc}(f,n)(z)(\mathtt{nil}))\ )\\
&\qquad\text{by } \textbf{(let.i)} \text{ and definition of } \kappa_n\\
&\quad\cong \mathtt{ncc}(\ \lambda k.(\ \mathtt{set}(z,k); \mathtt{top}(\mathtt{resume}(z)(f(n))); \mathtt{inc}(f,n+1)(z)(\mathtt{nil}))\ )\ )\\
&\qquad\text{using the definition of } \mathtt{inc}
\end{aligned}
$$

Next we unfold the `resume`, push the reduction context surrounding the `ncc` to the inside, drop the c that will be discarded by `top` and reduce the `app`.

$$
\begin{aligned}
&\mathtt{ncc}(\ \lambda k.(\ \mathtt{set}(z,k); \mathtt{top}(\mathtt{resume}(z)(f(n))); \mathtt{inc}(f,n+1)(z)(\mathtt{nil}))\ )\ )\\
&\quad\cong \mathtt{ncc}(\lambda k.(\ \mathtt{set}(z,k);\\
&\qquad\qquad \mathtt{top}(\mathtt{ncc}(\ \lambda k.\,\mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z,k); p(f(n))\ );\\
&\qquad\qquad\quad \mathtt{inc}(f,n+1)(z)(\mathtt{nil}))\ )\ )\\
&\qquad\text{by definition of } \mathtt{resume}(z)\\
&\quad\cong \mathtt{ncc}(\ \lambda k.(\ \mathtt{set}(z,k);\\
&\qquad\qquad \mathtt{ncc}(\ \lambda c.\mathtt{app}(\lambda k.\,\mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z,k); p(f(n))\ );\\
&\qquad\qquad\qquad \lambda x.c(\mathtt{top}(\mathtt{inc}(f,n+1)(z)(\mathtt{nil})))\ )\ )\ )\\
&\qquad\text{by } \textbf{(ncc.ii)} \text{ with } R = \mathtt{top}(\bullet; \mathtt{inc}(f,n+1)(z)(\mathtt{nil}))\\
&\quad\cong \mathtt{ncc}(\ \lambda k.(\ \mathtt{set}(z,k);\\
&\qquad\qquad \mathtt{ncc}(\ \lambda c.\,\mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z,\kappa_{n+1}); p(f(n))\ )\ )\ )\\
&\qquad\text{by the } \textbf{(top)} \text{ lemma and } \mathtt{let} \text{ laws}
\end{aligned}
$$

Finally, the `set` can be pushed inside the inner `ncc` and the result simplified using the memory and let laws.

$$
\begin{aligned}
&\quad\cong \mathtt{ncc}(\ \lambda k.(\ \mathtt{set}(z,k);\\
&\qquad\qquad \mathtt{ncc}(\ \lambda c.\,\mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z,\kappa_{n+1}); p(f(n))\ )\ )\ )\\
&\quad\cong \mathtt{ncc}(\ \lambda k.(\ \mathtt{ncc}(\ \lambda c.\mathtt{set}(z,k);\\
&\qquad\qquad\qquad \mathtt{let}\,\{p := \mathtt{get}(z)\}\mathtt{set}(z,\kappa_{n+1}); p(f(n))\ )\ )\ )\\
&\qquad\text{by } \textbf{(ncc.iv)}\\
&\quad\cong \mathtt{ncc}(\ \lambda k.\mathtt{ncc}(\ \lambda c.\mathtt{set}(z,\kappa_{n+1}); k(f(n))\ )\ )\\
&\qquad\text{by the } \mathtt{set} \text{ and } \mathtt{get} \text{ laws}
\end{aligned}
$$

Now we the desired ‘canonical’ form with directly nested ncc$s$. This allows us to further transform the body of the inner ncc and finally to eliminate it.

$$\begin{aligned}
&\texttt{ncc}(\ \lambda k.\texttt{ncc}(\ \lambda c.\texttt{set}(z, \kappa_{n+1}); k(f(n))\ )\ ) \\
&\quad\cong \texttt{ncc}(\ \lambda k.\texttt{app}(\lambda c.\texttt{set}(z, \kappa_{n+1}); k(f(n)), \texttt{top})\ ) \\
&\qquad\text{by the } (\textbf{ncc.i}) \\
&\quad\cong \texttt{ncc}(\ \lambda k.\texttt{set}(z, \kappa_{n+1}); k(f(n))\ ) \\
&\qquad\text{by the } (\textbf{let.i}) \\
&\quad\cong \texttt{set}(z, \kappa_{n+1}); \texttt{ncc}(\lambda k.k(f(n))) \\
&\qquad\text{by the } (\textbf{ncc.iv}) \\
&\quad\cong \texttt{set}(z, \kappa_{n+1}); k(f(n)) \\
&\qquad\text{by the } (\textbf{ncc.v})
\end{aligned}$$

This completes the proof. □

§**7. Conclusions and Future Directions.** In this paper we have shown how to generalize our previous variable-type theory [22] for reasoning about programs of a specific language Λ_{m} to a logic for reasoning about an arbitrary Landinesque language with uniform semantics and have given an example of axiomatizing a typical language of this class. The variable-type theories have the advantage of being essentially first-order, while providing for functions (operations) as first-class objects, and providing the ability to define and reason with classifications.

This is just the foundation for putting these ideas to work in practice. There are a number of interesting directions for future work. One direction is to consider a wider class of languages, for example treating object-oriented (OO) languages such as Java [20], or treating input/output (IO) features. We believe that the sequential aspects of OO languages are not problematic. Treating IO and concurrency aspects will be more challenging, as this means removing the restriction that the semantics be deterministic. It may be that the solution here is to combine the nice features of the variable-type logics with some form of temporal logic.

Another direction for future work is implementation—building tools to aid in the process of specifying and reasoning about programs using the Feferman–Landin logic. Here the work of Frost [17] would be a good starting point.

Although complete axiomatizations of Landinesque languages are not possible, it is useful to investigate forms of relative completeness and completenss for fragments such as done in [26] for the Λ_{m} language. This is especially important in conjunction with implementing the logic and developing mechanized procedures for aiding the development of proofs.

When reasoning informally about programs, we use a mix of operational and equational/logical reasoning. The importance of being able to combine these forms of reasoning was first noted in [19] where operationally based principles for denotational reasoning about Lisp programs were developed. An interesting and potentially very useful direction of future research in our context is to enrich the Feferman–Landin logic by formalizing the reduction relation, and adding a reflection principle that allows meta-theorems to be reflected down to the the logic level. For example special purpose simulation principles could be added in this way. Another application would be to extend the formal meta theory with definitions of program analysis functions and derive proof principles based on program analysis results. For example analysis could characterize certain classes of effects and principles generalizing the constraint propagation axioms for Λ_s could be developed in a formal setting. Again this mix of intensional and extensional (operational and denotational) reasoning is quite natural for programmers.

It may also be fruitful to investigate the use of other methods developed for reasoning about operational approximation and equivalence. These include: general schemes for establishing equivalence; context lemmas (alternative characterizations that reduce the number of contexts to be considered); and (bi)simulation relations (alternative characterizations or approximations based on co-inductively defined relations). For example [23] develops a schema for proving congruence for a class of languages with a particular style of operational semantics. This schema succeeds in capturing many simple functional programming language features. Building on this work, Howe [24] uses an approach similar to the idea of uniform computation to define structured evaluation systems in which the form of the evaluation rules guarantees that (bi)simulation relations are congruences. The form of the rules is specified using meta variables with arities and higher-order substitutions. This syntax enrichment is very similar to the notions of place-holder and filling used here to specify uniform semantics.

A useful refinement would be to identify a form of rules that guarantees uniform semantics, generalizing the ideas of Howe [24] to functional languages with effects. Another extension would be to develop denotational tools in this setting generalizing [28].

REFERENCES

[1] H. Abelson and G. J. Sussman, ***Structure and interpretation of computer programs***, The MIT Press, McGraw-Hill Book Company, 1985.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, *A foundation for actor computation*, ***Journal of Functional Programming***, vol. 7 (1997), pp. 1–72.

[3] K.R. Apt, *Ten years of Hoare's logic: A survey–part I*, ***ACM Transactions on Programming Languages and Systems***, vol. 4 (1981), pp. 431–483.

[4] C.C. Chang and H.J. Keisler, ***Model theory***, North-Holland, Amsterdam, 1973.

[5] S. Feferman, *A language and axioms for explicit mathematics*, ***Algebra and logic***, Springer Lecture Notes in Mathematics, vol. 450, Springer Verlag, 1975, pp. 87–139.

[6] ———, *Non-extensional type-free theories of partial operations and classifications, i.*, ***Proof theory symposium, kiel 1974*** (J. Diller and G. H. Müller, editors), Lecture notes in mathematics, no. 500, Springer, Berlin, 1975, pp. 73–118.

[7] ———, *Constructive theories of functions and classes*, ***Logic colloquium '78***, North-Holland, 1979, pp. 159–224.

[8] ———, *A theory of variable types*, ***Revista Colombiana de Matématicas***, vol. 19 (1985), pp. 95–105.

[9] ———, *Polymorphic typed lambda-calculi in a type-free axiomatic framework*, ***Logic and computation***, Contemporary Mathematics, vol. 106, A.M.S., Providence R. I., 1990, pp. 101–136.

[10] ———, *Logics for termination and correctness of functional programs,*, ***Logic from computer science***, MSRI Publications,, vol. 21, Springer Verlag, 1992, pp. 101–136.

[11] ———, *Logics for termination and correctness of functional programs, II. logics of strength PRA.*, ***Proof theory***, Cambridge University Press, 1992.

[12] M. FELLEISEN,, 1993, Personal communication.

[13] M. FELLEISEN and D.P. FRIEDMAN, *Control operators, the SECD-machine, and the λ-calculus*, ***Formal description of programming concepts III*** (M. Wirsing, editor), North-Holland, 1986, pp. 193–217.

[14] M. FELLEISEN and A. SABRY, *Continuations in programming practice: Introduction and survey*, 1999, available at `http://www.cs.uoregon.edu/~sabry/papers/continuations.ps`.

[15] J. FORD and I. A. MASON, *Establishing a General Context Lemma in PVS*, ***2nd Australasian Workshop on Computational Logic, AWCL'01***, 2001, available at `http://mcs.une.edu.au/~pvs/`.

[16] D. P. FRIEDMAN, *Applications of continuations*, ***Technical Report 237***, Indiana Univeristy Computer Science Department, 1988, Tutorial given at POPL88.

[17] J. FROST, *Effective programming*, ***Ph.D. thesis***, Technical University of Denmark, 1996, also published as Techincal Report IT-TR 1996-001.

[18] J. FROST and I. A. MASON, *An Operational Logic of Effects*, ***Proceedings of the Australasian Theory Symposium, CATS '96***, 1996, pp. 147–156.

[19] M. J. C. GORDON, *Operational reasoning and denotational semantics*, ***Technical Report SAIL Memo AIM-264***, Artifi cial Intelligence Laboratory, Stanford University, 1975.

[20] J. GOSLING, B. JOY, and G. L. STEELE JR., ***The java language specification***, Addison-Wesley, 1996.

[21] D. HAREL, *Dynamic logic*, ***Handbook of philosophical logic, vol. ii*** (D. Gabbay and G. Guenthner, editors), D. Reidel, 1984, pp. 497–604.

[22] F. HONSELL, I. A. MASON, S. F. SMITH, and C. L. TALCOTT, *A Variable Typed Logic of Effects*, ***Information and Computation***, vol. 119 (1995), no. 1, pp. 55–90.

[23] D. HOWE, *Equality in the lazy lambda calculus*, ***Fourth annual symposium on logic in computer science***, IEEE, 1989.

[24] D. J. HOWE, *Proving congruence of bisimulation in functional programming languages*, ***Information and Computation***, vol. 124 (1996), no. 2, pp. 103–112.

[25] P. J. LANDIN, *The mechanical evaluation of expressions*, ***Computer Journal***, vol. 6 (1964), pp. 308–320.

[26] I. A. MASON, *A First Order Logic of Effects*, ***Theoretical Computer Science***, vol. 185 (1997), pp. 277 – 318.

[27] ———, *Computing with contexts*, ***Higher-Order and Symbolic Computation***, vol. 12 (1999), pp. 171–201.

[28] I. A. MASON, S. F. SMITH, and C. L. TALCOTT, *From Operational Semantics to Domain Theory*, ***Information and Computation***, vol. 128 (1996), no. 1, pp. 26–47.

[29] I. A. MASON and C. L. TALCOTT, *Equivalence in functional languages with effects*, ***Journal of Functional Programming***, vol. 1 (1991), pp. 287–327.

[30] ———, *Reasoning about object systems in VTLoE*, ***International Journal of Foundations of Computer Science***, vol. 6 (1995), no. 3, pp. 265–298.

[31] R. MILNER, *Fully abstract models of typed λ-calculi*, ***Theoretical Computer Science***, vol. 4 (1977), pp. 1–22.

[32] R. MILNER, M. TOFTE, and R. HARPER, ***The definition of standard ML***, MIT Press, 1990.

[33] E. MOGGI, *Computational lambda-calculus and monads*, ***Fourth annual symposium on logic in computer science***, IEEE, 1989.

[34] A. M. PITTS, *Evaluation logic*, ***Ivth higher-order workshop, banff***, Workshops in Computing, vol. 283, Springer-Verlag, 1990.

[35] G. PLOTKIN, *Call-by-name, call-by-value and the lambda calculus*, ***Theoretical Computer Science***, vol. 1 (1975), pp. 125–159.

[36] D. PRAWITZ, ***Natural deduction: A proof-theoretical study***, Almquist and Wiksell, 1965.

[37] J.C. REYNOLDS, *Idealized ALGOL and its specification logic*, ***Tools and notions for program construction*** (D. Néel, editor), Cambridge University Press, 1982, pp. 121–161.

[38] G. L. STEELE and G. J. SUSSMAN, *Scheme: An interpreter for extended lambda calculus*, ***Higher-Order and Symbolic Computation***, vol. 11 (1999), no. 4, pp. 405–439.

[39] C. L. TALCOTT, *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*, ***Ph.D. thesis***, Stanford University, 1985.

[40] ———, *A theory for program and data type specification*, ***Theoretical Computer Science***, vol. 104 (1992), pp. 129–159.

[41] ———, *A theory for program and data specification*, ***Theoretical Computer Science***, vol. 104 (1993), pp. 129–159.

[42] ———, *Reasoning about functions with effects*, ***Higher order operational techniques in semantics***, Cambridge University Press, 1996.

SCHOOL OF MATHEMATICS AND COMPUTER SCIENCE
UNIVERSITY OF NEW ENGLAND
ARMIDALE 2351, N.S.W, AUSTRALIA
E-mail: iam@turing.une.edu.au

DEPARTMENT OF COMPUTER SCIENCE
STANFORD UNIVERSITY
STANFORD, CA 94305, USA
E-mail: clt@cs.stanford.edu