

Reasoning about Functions with Effects

Carolyn Talcott

Department of Computer Science

Stanford University

clt@sail.stanford.edu

<http://www-formal.stanford.edu/clt/home.html>

1 Introduction

An important advantage claimed for functional programming is the elegant equational theory and the ability to reason about programs by simply replacing equals by equals. Also higher-order functions provide the ability to define high-level abstractions that simplify programming. On the other hand, real programs have effects, and it is often more concise to express a computation using effects. We would like to have the best of both worlds: functions with effects. The methods described in this paper allow one to have the expressive power of functions with effects and to develop rich equational theories for such languages. We extend our previous work on semantics of imperative functional languages (Talcott 1985, 1992; Mason and Talcott 1991a, 1992; Honsell, Mason, Smith, and Talcott 1995) by treating the combination of control and memory effects, and developing the semantic theory in a more abstract setting to better capture the essential features of program equivalence.

We adopt the view proposed in (Landin 1966) that a programming language consists of expressions of the lambda calculus augmented with primitive operations. We call such languages λ -languages. The primitive operations that we have in mind include not only basic constants, branching, and algebraic operations such as arithmetic and pairing, but also operations that manipulate the computation state (store, continuation), and the environment (sending messages, creating processes). The presence of higher-order objects in λ -languages makes defining and reasoning about program equivalence more complicated than in the first-order case, with or without effects. The methods and results described in this paper arose from a desire to treat a wide range of λ -languages in a unified manner. Although we only consider sequential languages here, the key ideas seem to extend to primitives for concurrency as well (Agha, Mason, Smith, and Talcott 1997).

Overview

We take an operational approach to defining the semantics of λ -languages. We begin 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 in 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.

We then abstract from the details of computations by defining the notions of operational approximation and equivalence on expressions. Intuitively, two expressions are operationally equivalent (also called contextually equivalent in the literature) if no program context can distinguish them. More precisely, operational equivalence is the equivalence naturally associated with the operational approximation pre-order. One expression operationally approximates another expression if when placed in any closing program context either the resulting first program is undefined or both resulting programs are defined.

Some laws of program equivalence, including laws that axiomatize operations with effects, hold because of the general nature of the languages we are considering. These laws are robust in the sense that they are not invalidated by addition of new primitive operations to the language (subject to minimal constraints on the semantics). They include the laws of computational lambda calculus (Moggi 1988), and the program calculi developed in (Felleisen 1987; Felleisen and Hieb 1992) for Scheme-like languages. These laws typically have simple computational justifications. For example, two expressions are equivalent if they reduce to a common expression (while having the same effect on the computation state). As another example, two expressions are equivalent if they correspond to placing a third expression in computationally equivalent contexts. Other laws, such as those that change the order of evaluation, need more careful formulation as they are sensitive to the particular choice of primitive operations. In order to get a better understanding of these observations, we identify properties of the operational semantics of a λ -language — structure of computation states and reduction rules — which allow us to establish computational laws in a quite general setting. If these properties hold we say that the language has *uniform semantics*. This allows us to build an equational core that holds for all λ -languages with uniform semantics, as well as to investigate more specialized laws in the same framework.

A key result for λ -languages with uniform semantics is an alternate characterization of the operational approximation relation that reduces the number of contexts that must be considered to establish laws of approximation and equivalence. In particular we show that we only need to consider contexts

which correspond to computation states in which the hole corresponds to the program counter (the expression to be evaluated next). We call this the **ciu** theorem since the contexts considered correspond to all Closed Instantiations of Uses of the expressions to be tested. Although rudimentary, this characterization turns out to be quite useful. Our method for establishing the correctness of this alternative characterization relies on the notion of uniform computation, which underlies the definition of uniform semantics. Uniform computation allows computation steps to be carried out on states with missing information. It has the property that computing commutes with filling in of missing information. Using **ciu** and other consequences of uniform semantics we then establish some general principles for proving program equivalence that capture the computational intuitions. To make these ideas concrete, we give uniform semantics and derive equivalence laws for algebraic, control, and memory operations which combined form the kernel of a Scheme- or (untyped) ML- like language.

Related Work

Previous work of Talcott, Mason, Felleisen, and Moggi establishes a mathematical foundation for studying notions of program equivalence for programming languages with function and control abstractions operating on objects with memory. This work builds on work of Landin, Reynolds, Morris and Plotkin. Landin (1964) and Reynolds (1972) describe high-level abstract machines for defining language semantics. Morris (1968) defines an extensional equivalence relation for the classical lambda calculus. Plotkin (1975) extends these ideas to the call-by-value lambda calculus and introduces the operational equivalence relation.

Talcott (1985); Mason (1986, 1988); Talcott (1989); Mason and Talcott (1991a, 1992) develop various operational methods for the studying operational approximation and equivalence for subsets of a language with function and control abstractions and objects with memory. Agha, Mason, Smith, and Talcott (1992, 1997) extend these methods to develop an operational semantics of a λ -language with primitives for actor computation (distributed object-based computation). Felleisen (1987) studies reduction calculi extending the call-by-value lambda calculus to languages with control and assignment abstractions. These calculi are simplified and extended in (Felleisen and Hieb 1989). The notion of computational monad as a framework for axiomatizing features of programming languages is introduced in (Moggi 1989, 1990). Reduction calculi and operational equivalence both provide a sound basis for purely equational reasoning about programs. Calculi have the advantage that the reduction relations are inductively generated from primitive reductions (such as beta-conversion) by closure operations (such as transitive closure or congruence closure). Equations proved in a calculus continue to hold when the calculus is extended to treat additional language constructs.

Operational equivalence is, by definition, sensitive to the set of language constructs and basic data available. However, pure reduction calculi are not adequate to prove many basic equivalences in languages with effects. For example, Felleisen found it is necessary to extend his reduction calculus by meta principles (cf. the **safety rule** of Felleisen (1987), thm 5.27, p.149). Using operational approximation we can express and prove properties such as non-termination, computation induction and existence of least fixed points which cannot even be expressed in reduction calculi. The uniform semantics framework presented in this paper provides the extensibility of reduction calculi for a wide range of λ -languages. Operations are given semantics by reduction rules that are not modified when new features are added to the language. This kind of modularity is in the spirit of action semantics (Mosses 1992). The objective of action semantics is to support modular (compositional) specification of the semantics of a language allowing each feature to be specified independently, and allowing new features to be added without modifying the specification of the original language. This is accomplished by splitting computation state into orthogonal facets and defining a set of action combinators to be used as denotations. Action semantics treats a wider range of languages, and provides a number of generic tools. However, development of equational theories derived from action semantics has only recently begun (Lassen 1995b) (see below).

The fact that one can present a syntactic reduction system for imperative λ -calculi was discovered independently in 1986-1987: by Talcott (Mason and Talcott 1991a), and by Felleisen and Hieb (1992). As well as being conceptually elegant, it has also provided the necessary tools for several key results and proofs. In addition to eliminating messy isomorphism considerations, to deal with arbitrary choice of names of newly allocated structures, it also was a key step leading to the formulation of the **ciu** theorem, first presented in (Mason and Talcott 1989). In 1987, Mason realized that it provided the ideal notion of a normal form and symbolic evaluation needed in the completeness result presented in (Mason and Talcott 1992). Syntactic reduction systems provided the basis for the elegant revision of the imperative calculi of Felleisen (1987) that was published in (Felleisen and Hieb 1992). Other successful uses of the technique include: the type soundness proof, via subject reduction, for the imperative ML type system (Felleisen and Wright 1991); the analysis of parameter passing in Algol (Crank and Felleisen 1991; Weeks and Felleisen 1993); and the analysis of reduction calculi for Scheme-like languages (Sabry and Felleisen 1993; Fields and Sabry 1993).

Much work has been done to develop methods for reasoning about operational approximation and equivalence: Abramsky (1990, 1991); Bloom (1990); Egidi, Honsell, and Ronchi della Rocca (1992); Howe (1989, 1996); Gordon (1995); Lassen (1995b); Mason (1986); Mason and Talcott (1991a); Jim and Meyer (1991); Milner (1977); Ong (1988); Pitts and Stark (1993, 1996); Ritter and Pitts (1995); Pitts (1996); Smith (1992); Sullivan (1996); Talcott

(1985). Methods developed for reasoning about operational approximation and equivalence 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). An early example is Milner's context lemma (Milner 1977) which greatly simplifies the proof of operational equivalence in the case of the typed lambda calculus by reducing the contexts to be considered to a simple chain of applications. Talcott (1985) studies general notions of equivalence for languages based on the call-by-value lambda calculus, and develops several schemes for establishing properties of such relations. Howe (1989) 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 (1996) 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. The idea of using such meta terms to specify classes of rules giving rise to reduction relations with special properties was used in (Aczel 1978) to prove a general Church-Rosser theorem and in (Klop 1980) to develop the theory of Combinatory Reduction Systems. Meta terms are also used in describing a unification procedure for higher-order patterns in (Nipkow 1991). As discussed above, Mason and Talcott (1989, 1991a) introduced the **ciu** characterization of operational equivalence which is a form of context lemma for imperative languages. The uniform computation method used to first prove **ciu** was adapted to develop computational path transformation methods for proving equivalence of actor programs in (Agha, Mason, Smith, and Talcott 1997). Ritter and Pitts (1995) use operational techniques to establish the correctness of a translation between an imperative subset of standard ML and a simply typed lambda calculus with reference types. An applicative bisimulation is defined and shown to be sound for operational equivalence. This relation is adequate for establishing the correctness of the translations in question, but is weaker than operational equivalence. Finding a bisimulation relation that coincides with operational equivalence in the case of imperative higher-order languages such as Scheme or ML remains an open problem (to this author's knowledge). Pitts (1996) proves a context lemma for a higher-order language with assignable variables that only store first-order values. The proof uses logical relations that are defined in terms of the operational semantics. The logical relation mechanism combined with the context lemma provide a useful method for establishing program equivalence. Pitts and Stark (1996) extend these ideas to a language with richer types. A challenging problem is to apply the logical relations approach to un-

typed languages. Sullivan (1996) also takes a mixed operational-denotational approach. A metalanguage based on PCF extended with I/O and dynamic store primitives is defined operationally. A context lemma for this language is proved by proving that an applicative simulation relation is a precongruence. One can reason about programs with imperative features by giving them compositional denotations in this metalanguage. This is promising blend of operational and denotational semantics, providing a better approximation to operational equivalence than most existing denotational approaches. Gordon (1995) uses standard process algebra techniques to derive bisimulation relations from labelled transition systems based on operational semantics. In the case of typed functional languages (PCF+streams) bisimulations can be found that coincide with operational equivalence. It will be interesting to see if this approach to defining bisimulations can be extended to imperative λ -languages, to develop useful if not complete reasoning tools. Lassen (1995b) presents an approach to developing a general framework for reasoning about program equivalence based on action semantics (Mosses 1992). Several operational pre-orderings are defined for a portion of action notation adequate for functional languages. Although there are no imperative effects, the action semantics allows for non-deterministic primitives, thus the interest in multiple pre-orders. Simulation relations that are essentially complete are defined for each of the pre-orders and used to verify properties of an (untyped) PCF-like language by giving the language an action semantics. A next stage in this effort is to consider actions that support imperative primitives. A co-induction rule for establishing operational equivalence in a λ -language with reference cells that captures much of the reasoning in the Mason-Talcott papers based on computation induction and the **ciu** theorem has been suggested by S. Lassen (1995a).

Plan

The remainder of the paper is organized as follows. In §2. we develop the general framework for studying equivalence of programs in λ -languages. The basic syntactic notions are introduced, the semantic notions of definedness and equivalence are defined (relative to details to be filled in for specific languages), the notion of uniform semantics is introduced and the key properties of languages with uniform semantics are stated. In §3. we give the semantics for a representative collection of functional primitives, and discuss equational laws for the functional primitives valid in any λ -language with uniform semantics. We also consider some properties specific to the functional language. In §4. we give the semantics for a typical control primitive and discuss its equational theory. In §5. we give the semantics for a collection of primitives for allocating, accessing and updating ML-like reference cells and discuss the equational theory for these operations. In §6. we combine the primitives of the previous section into a full Scheme-like language and discuss

the ramifications to the individual equational theories. In §7. we introduce the place-holder machinery needed to fill in the details in the definition of uniform semantics for λ -languages. We then show that the languages defined in the previous sections have uniform semantics. In §8. we use uniform computation techniques to establish the **ciu** theorem for λ -languages with uniform semantics. We then establish several other results that provide a basis for developing a core equational theory valid for all such languages. §9. contains some concluding remarks.

Along the way we give some very simple programming and proving examples to provide basic intuitions about the various program primitives and reasoning principles. Many more examples of programming and proving with functions, control, and memory can be found in (Burge 1975); (Talcott 1985 1989, 1992); (Felleisen 1987, 1988); (Mason 1986); (Mason and Talcott 1990, 1991a, 1991b, 1992, 1994a, 1994b). Examples include: higher-order functionals as generic program modules; manipulating mutable lists; stream processing; co-routines; and objects as functions (lambda abstractions) with state.

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, \dots , range over the set X . $\text{Fmap}[X_0, X_1]$ is the set of finite maps from X_0 to X_1 . We write $\text{Dom}(f)$ for the domain of a function and $\text{Rng}(f)$ for its range. For any function f , $f\{x \mapsto x'\}$ is the function f' such that $\text{Dom}(f') = \text{Dom}(f) \cup \{x\}$, $f'(x) = x'$, and $f'(z) = f(z)$ for $z \neq x, z \in \text{Dom}(f)$. Also $f[X$ is the restriction of f to X : the function f' such that $\text{Dom}(f') = \text{Dom}(f) \cap X$ and $f'(x) = f(x)$ for $x \in \text{Dom}(f)$. $\mathbf{N} = \{0, 1, 2, \dots\}$ is the set of natural numbers and i, j, n, n_0, \dots 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:

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

2 The General Framework

In this section a general framework for studying the semantics of λ -languages is set up. 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. To simplify reasoning about effects we restrict attention to call-by-value semantics, but we see no problem is adapting the basic ideas to other evaluation strategies.

2.1 Expression Syntax of a λ -language

Fix a countably infinite set of variables, \mathbf{X} . The basic syntax of a λ -language is then determined by giving a countable set of atoms, \mathbf{A} , and 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) such that the sets \mathbf{X} , \mathbf{A} , \mathbf{O}_n for $n \in \mathbf{N}$ are pairwise disjoint. We assume that \mathbf{O} contains at least the binary operation **app** (lambda application). For example, taking $\mathbf{A} = \{ \}$ and $\mathbf{O} = \{\mathbf{app}\}$ we obtain the expressions of the pure call-by-value lambda calculus, Λ_v .

Definition (\mathbf{E}, \mathbf{L}): The set of expressions, \mathbf{E} , and the set of λ -abstractions, \mathbf{L} , are defined as the least sets satisfying the following equations:

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

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

We let a range over \mathbf{A} , x, y, z range over \mathbf{X} , e range over \mathbf{E} , and φ range over \mathbf{L} . Elements of \mathbf{L} are called *lambda abstractions* or more briefly *lambdas*. λ is a binding operator with free and bound variables of expressions defined as usual. Two expressions are considered equal if they are the same up to renaming of bound variables. $\text{FV}(e)$ is the set of free variables of e , and we write $\text{FV}(e_1, \dots, e_n)$ for $\text{FV}(e_1) \cup \dots \cup \text{FV}(e_n)$. A *closed expression* is an expression with no free variables. Substitution $e^{\{x \mapsto e_0\}}$ of e_0 for free occurrences of x in e is defined as usual, renaming bound variables of e where needed to avoid capture of free variables of e_0 . We will make use of the following common abbreviations and notation conventions.

$$\lambda x_1, \dots, x_n. e \triangleq \lambda x_1. \dots \lambda x_n. e$$

$$\mathbf{app}(e_0, e_1, \dots, e_n) \triangleq \mathbf{app}(\dots \mathbf{app}(e_0, e_1), \dots e_n)$$

$$e_0(e_1, \dots, e_n) \triangleq \mathbf{app}(e_0, e_1, \dots, e_n)$$

$$\mathbf{let } x = e_0 \mathbf{ in } e_1 \triangleq \mathbf{app}(\lambda x. e_1, e_0)$$

$$v_0 \circ v_1 \triangleq \lambda x. v_0(v_1(x)) \quad \text{if } x \notin \text{FV}(v_0, v_1)$$

$$e_0; e_1 \triangleq \mathbf{let } d = e_0 \mathbf{ in } e_1 \quad \text{sequencing where } d \notin \text{FV}(e_0, e_1)$$

$$\mathbf{Y}_v \triangleq \lambda f. \mathbf{let } h = \lambda h. \lambda x. \mathbf{app}(\mathbf{app}(f, \mathbf{app}(h, h)), x) \mathbf{ in } \mathbf{app}(h, h)$$

call-by-value recursion combinator

2.2 Operational Semantics – Overview

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. 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, including specifying
 - the empty state context
 - the map giving the expression associated to a state
- the reduction rule for each primitive operation

2.3 Operational Semantics – Details

We now make precise the concepts discussed above. As pointed out above, some features are fully defined (relative to others), while others must obey certain constraints, but may vary depending on the particular choice of language. To distinguish these situations, we use the header **Definition** to signal definitions of uniformly defined features and the header **Specification** to signal constraints on language dependent features.

We begin with the concepts of context, value expression, and value substitution needed to state the definitions of definedness and operational equivalence. These are also sufficient to state the first requirement for uniform semantics, **g-unif** (global uniformity). We then introduce additional notions

of reduction context and redex. This allows us to describe the form of computation rules and to state informally the second requirement for uniform semantics, **s-unif** (stepwise uniformity). Finally we state the **ciu** theorem and some consequences that formalize the principles underlying the computational laws of equivalence.

Definition (Contexts (C)): Contexts are expressions with holes. We use \bullet to denote a hole. The set of contexts, \mathbf{C} , is defined by

$$\mathbf{C} = \{\bullet\} \cup \mathbf{X} \cup \mathbf{A} \cup \lambda\mathbf{X}.\mathbf{C} \cup \mathbf{O}_n(\mathbf{C}^n)$$

We let C range over \mathbf{C} . $C[\epsilon]$ denotes the result of replacing each hole in C by ϵ . Free variables of ϵ may become bound in this process. For example free occurrences of x will become bound when an expression is placed in the hole in $\lambda x.\bullet$, as in $(\lambda x.\bullet)[x] = \lambda x.x$. We let $\text{Traps}(C)$ be the set of lambda variables of C with a hole in their scope – the variables that can be trapped when the holes are filled. For example, $\text{Traps}(\lambda x.\bullet) = \{x\}$. Note that renaming of bound variables in a context is not allowed since it changes the meaning of the context. For example, $(\lambda z.\bullet)[x] = \lambda z.x \neq \lambda x.x$. Many contexts of interest have the property that there are no holes in the scope of a λ , and for such contexts, renaming of bound variables is still valid, and we extend the application of substitutions to such contexts by defining $\bullet^{\{x \mapsto \epsilon\}} = \bullet$.

Specification (Value Expressions (V), Value Substitutions (S)):

The set of value expressions, \mathbf{V} , contains all variables, atoms, and lambdas. It may in addition contain expressions of the form $\vartheta(v^n)$. Value substitutions, \mathbf{S} , are finite maps from variables to value expressions. \mathbf{V} must be closed under application of value substitutions. More precisely, we require:

$$\mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \subseteq \mathbf{V} \subseteq \mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{O}_n(\mathbf{V}^n)$$

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

$$\mathbf{V}^{\mathbf{S}} = \mathbf{V}$$

We let v range over \mathbf{V} and σ range over \mathbf{S} . e^σ is the result of simultaneous substitution of free occurrences of $x \in \text{Dom}(\sigma)$ in e by $\sigma(x)$, taking care not to trap variables. 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 $i < n$. 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.

Specification (State contexts (Z)): \mathbf{Z} is the subset of \mathbf{C} consisting of the set of state contexts, ζ ranges over \mathbf{Z} and $\diamond \in \mathbf{Z}$ is the empty state context. ζ^σ is the result of ‘replacing’ free occurrences of $x \in \text{Dom}(\sigma) - \text{Traps}(\zeta)$ by $\sigma(x)$. Variables in $\text{FV}(\sigma) \cap \text{Traps}(\zeta)$ may become bound in the process.

The full determination of \mathbf{Z} , \diamond , and ζ^σ must be made for each λ -language as a part of defining its semantics. In the languages considered here, the

state context will be a context with a single hole. The context surrounding the hole can be thought of as a canonical form describing the state in which an expression placed in that hole is to be evaluated. The variables trapped at the hole in a state context provide a means of naming dynamically created values. For example, if there are no operators with effects, as in Λ_v , then $\diamond = \bullet$ and $\mathbf{Z} = \{\diamond\}$. In a language with a primitive operation, **aref**, that constructs reference cells for atoms, $\zeta^{\text{aref}} = \text{let } z_1 = \text{aref}(a_1) \text{ in let } z_2 = \text{aref}(a_2) \text{ in } \bullet$ would be a state context representing a state with two reference cells that can be referred to as z_1 and z_2 . z_1 initially contains a_1 and z_2 initially contains a_2 . As we will see in §5, slightly more complex contexts are needed to represent memory that may contain cycles. The states of languages considered here can be decomposed as $\zeta = \zeta_a[\zeta_e]$ where all trapping is done in ζ_a (a for allocation) and ζ_e describes the state of the allocated structures (e for effects). In this case we define $\zeta^\sigma = \zeta_a[\zeta_e^\sigma]$ for $\text{Dom}(\sigma) \cap \text{Traps}(\zeta) = \emptyset$. Continuing the example of reference cells for atoms, assume there is an atom **nil**, and an assignment operation $z := v$ that assigns v to the cell z when v is an atom. Then we refine state contexts so that ζ^{aref} becomes $\zeta_a^{\text{aref}}[\zeta_e^{\text{aref}}]$ where $\zeta_a^{\text{aref}} = \text{let } z_1 = \text{aref}(\text{nil}) \text{ in let } z_2 = \text{aref}(\text{nil}) \text{ in } \bullet$ and $\zeta_e^{\text{aref}} = z_1 := a_1; z_2 := a_2; \bullet$.

Specification (Computation States (CS)): $\mathbf{CS} = \mathbf{Z} : \mathbf{E}$ is the set computation states. We let S range over \mathbf{CS} and let $\zeta : e$ be the state with state context ζ and expression e . $\text{s2e}(_)$ is the map associating to each state a representing expression. We will restrict attention to the case when the mapping is defined by hole filling: $\text{s2e}(\zeta : e) = \zeta[e]$. A state is *closed* just if its corresponding expression is closed. Application of value substitutions to states is defined by: $(\zeta : e)^\sigma = \zeta^\sigma : e^\sigma$ for $\text{Dom}(\sigma) \cap \text{Traps}(\zeta) = \emptyset$.

Specification (Reduction (\longrightarrow , \longrightarrow^*)): \longrightarrow is the single step reduction relation on states and \longrightarrow^* is the reflexive transitive closure of \longrightarrow .

As an example the single step reduction relation in Λ_v is the least relation such that

- : $\text{app}(\lambda x.e, v) \longrightarrow \bullet : e^{\{x \mapsto v\}}$
- : $e \longrightarrow \bullet : e' \Rightarrow \bullet : \text{app}(e, e_1) \longrightarrow \bullet : \text{app}(e', e_1)$
- : $e \longrightarrow \bullet : e' \Rightarrow \bullet : \text{app}(v, e) \longrightarrow \bullet : \text{app}(v, e')$

We will see below how to define the rule for **app** for an arbitrary λ -language. Notice that we do not restrict the reduction relation to closed states.

Definition (Definedness): For states S, S_0, S_1 , definedness, $S \downarrow$, approximation, $S_0 \preceq S_1$, equi-definedness, $S_0 \uparrow S_1$, and computation length of defined states, $|S|$, are defined by

$$S \downarrow \Leftrightarrow (\exists v \in \mathbf{V}, \zeta \in \mathbf{Z})(S \longrightarrow^* \zeta : v)$$

$$S_0 \preceq S_1 \Leftrightarrow (S_0 \downarrow \Rightarrow S_1 \downarrow)$$

$$S_0 \Downarrow S_1 \Leftrightarrow (S_0 \Downarrow \Leftrightarrow S_1 \Downarrow)$$

$|S|$ is the least $n \in \mathbf{N}$ such that S reduces to a value state in n steps, if $S \Downarrow$.

The first set of requirements for uniform semantics can now be stated. 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 (Global uniformity (g-unif)): A λ -language satisfies **g-unif** if the following hold.

$$\text{(unicity)} \quad S \longrightarrow S_0 \wedge S \longrightarrow S_1 \Rightarrow \text{s2e}(S_0) = \text{s2e}(S_1)$$

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

if $\text{Dom}(\sigma) \cap (\text{Traps}(\zeta) \cup \text{Traps}(\zeta')) = \emptyset$

$$\text{(rep)} \quad \diamond : \text{s2e}(S) \Downarrow S$$

$$\text{(red)} \quad S \longrightarrow S' \wedge S \Downarrow \Rightarrow S' \Downarrow \wedge |S'| < |S|$$

In the languages we consider **(unicity)** holds because the only non-determinism in a reduction step is the choice of names used in the state context. **(rep)** holds because reduction of $\diamond : \text{s2e}(\zeta : e)$ essentially recreates the state context ζ . **(red)** says that if a state is defined, then any reduction makes progress. Clearly if the reduct state is defined, then the original state is defined. It is easy to see that Λ_v satisfies these properties. The only hard part is to show that value substitution and beta-v reduction commute, which is a standard result.

2.4 Approximation and Equivalence

Now we define operational approximation and equivalence and lay the ground work for studying properties of these relations.

Definition (Approximation $e_0 \sqsubseteq e_1$, Equivalence $e_0 \cong e_1$):

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall C \mid \text{FV}(C[e_0], C[e_1]) = \emptyset) (\diamond : C[e_0] \preceq \diamond : C[e_1])$$

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

It is easy to see that operational approximation is a congruence: if $e_0 \sqsubseteq e_1$, then $C[e_0] \sqsubseteq C[e_1]$. Similarly for operational equivalence.

Two simple examples which can be stated in Λ_v are

$$\text{(lv.1)} \quad \text{let } x = (\text{let } y = e_y \text{ in } e_x) \text{ in } e \cong \text{let } y = e_y \text{ in let } x = e_x \text{ in } e$$

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

$$\text{(lv.2)} \quad \text{app}(\lambda x. \text{app}(x, x), \lambda x. \text{app}(x, x)) \sqsubseteq e$$

(**lv.1**) is an example of an expression, e_y , placed in two computationally equivalent contexts. This will be made precise below. (**lv.2**) holds because an expression that is not defined in any context, approximates any other expression. We will see below how to establish these results for an arbitrary λ -language with uniform semantics.

To define reduction rules and to state additional 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 $\vartheta(v_1, \dots, v_n)$ may be value expressions. For example, $\mathbf{app}(\lambda x.e, v)$ is a redex in any λ -language, while $\mathbf{pr}(v_0, v_1)$ is not a redex in any λ -language whose operations contain the binary pairing constructor, \mathbf{pr} .

Definition (Redexes (\mathbf{E}_r)): The set of redexes, \mathbf{E}_r , is defined by:

$$\mathbf{E}_r = \mathbf{O}_n(\mathbf{V}^n) - \mathbf{V}$$

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 correspond to the left-first, call-by-value reduction strategy of Plotkin (1975) and were first introduced by Felleisen and Friedman (1986).

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

$$\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{m+n+1}(\mathbf{V}^m, \mathbf{R}, \mathbf{E}^n)$$

We let R range over \mathbf{R} . $\mathbf{app}(\bullet, e)$ is a reduction context for any e , and $\mathbf{app}(v, \bullet)$ is a reduction context for any v , but neither $\mathbf{app}(\mathbf{app}(v_0, v_1), \bullet)$ nor $\lambda x.\mathbf{app}(\bullet, e)$ are reduction contexts. Since the hole of a reduction context is not in the scope of any bound variables, no free variables are trapped when filling the hole of a reduction context. In particular the application of value substitutions extends to reduction contexts.

It is easy to check that an expression is either a value expression or decomposes uniquely into a redex placed in a reduction context (a proof can be found in (Mason and Talcott 1991a)).

Lemma (Decomposition): If $e \in \mathbf{E}$ then either $e \in \mathbf{V}$ or e can be written uniquely as $R[r]$ where R is a reduction context and $r \in \mathbf{E}_r$.

For the languages considered here, the single step reduction relation is defined by giving a reduction rule for each operation. Ideally, we would like to give the rule for an operation, independent of the language, subject to constraints ensuring that state contexts are adequate to support the operation. This is a difficult task. Our approximate solution is to define the reduction rules using minimal information about the context $\zeta : R$ surrounding a redex. More detail is given in §3. As a first simple example, the rule for the

application operation **app** can be expressed in a language independent way as follows:

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

Now we consider the remaining requirements for a uniform semantics, **s-unif**. The objective is to formalize the requirement that each reduction rule is uniform in the form of its parameters. A reduction step may depend on the kind of construction of a redex argument, but not on any information about subparts. We do this by enriching the syntax of λ -languages to include place-holders for various syntactic sorts. In §7. we give the details of the syntax enrichment, and show how the reduction rules can be lifted to states of the enriched syntax. Here we introduce just enough notation to aide in reading the stepwise requirement. We decorate metavariables with \star to signify the enriched forms. Thus $\star e$ is an expression of the enriched syntax. $\star e[\star \mapsto e_0]$ is the result of filling expression place-holders in $\star e$ with e_0 . Similar notation is used for filling place-holders of other sorts in entities of other sorts.

Definition (Uniform reduction (s-unif)):

A λ -language satisfies **s-unif** if the single step reduction relation can be lifted to states of the enriched syntax so that:

1. If $\star \zeta : \star e \longrightarrow \star \zeta_1 : \star e_1$, then $(\star \zeta : \star e)[\star \mapsto x] \longrightarrow (\star \zeta_1 : \star e_1)[\star \mapsto x]$ for any x of a sort for which there are place-holders.
2. If $(\star \zeta : \star e)[\star \mapsto x] \longrightarrow \star \zeta' : \star e'$ then either $\star \zeta : \star e$ touches a hole ($\star e$ has the form $\star R[P]$ for some place-holder P), or $\star \zeta : \star e \longrightarrow \star \zeta_1 : \star e_1$, for some $\star \zeta_1 : \star e_1$ such that $\star \zeta' : \star e' = (\star \zeta_1 : \star e_1)[\star \mapsto x]$.

Definition (Uniform semantics): A λ -language has *uniform semantics* if it satisfies **g-unif** and **s-unif**.

A key result for λ -languages with uniform semantics is the **ciu** theorem. This theorem reduces the number of contexts that need to be considered when establishing operational approximation and equivalence. **ciu** is a form of context lemma (Milner 1977). Typically context lemmas characterize equivalence using only applicative contexts – contexts A of the form $A = \bullet$ or $A = \mathbf{app}(A', v)$ – observing termination and equality of observable values (booleans, numbers, etc.). Our context lemma uses arbitrary reduction contexts, observing termination only. To state the **ciu** theorem we introduce the **ciu**-approximation relation which holds just if each Closed Instantiation of a Use of the first expression approximates (as a state) the same Closed Instantiation of a Use of the second expression.

Definition (ciu-approximation $e_0 \sqsubseteq^{\text{ciu}} e_1$):

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

Theorem (ciu): For a λ -language with uniform semantics operational approximation and **ciu**-approximation coincide:

$$e_0 \sqsubseteq e_1 \Leftrightarrow e_0 \sqsubseteq^{\text{ciu}} e_1$$

The proof of this theorem will be given in §8 using the uniform computation machinery developed in §7. The approximation (**lv.2**) above is an easy consequence of **ciu**. Another consequence is the following.

Corollary (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.

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

Definition (Context independent reduction ($\xrightarrow{\text{ci}}$)):

A redex $\vartheta(v_1, \dots, v_n)$ is CI if the interpretation is independent of the context: either there is no reduction possible in any state, or the redex is replaced by the same reduct expression in any state. That is, exactly one of the following holds:

1. For any ζ, R there is no S' such that $\zeta : R[\vartheta(v_1, \dots, v_n)] \longrightarrow S'$.
2. There is some e such that $\zeta : R[\vartheta(v_1, \dots, v_n)] \longrightarrow \zeta : R[e]$ 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 $e \xrightarrow{\text{ci}} e'$ if $\zeta : R[e] \longrightarrow \zeta : R[e']$ 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 (**equi-reduct**). Similarly reasoning that two expressions are equivalent if the result from placing a third expression computationally equivalent reduction contexts is justified by (**equi-rcx**).

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

Proof : Assume $e_j \xrightarrow{\text{ci}} e$ for $j < 2$. By **ciu** (and symmetry), to show that $e_0 \cong e_1$ we need only show $e_0 \sqsubseteq^{\text{ciu}} e_1$. Pick some closing ζ, R, σ and assume $\zeta : R[e_0^\sigma] \downarrow$. By assumption and **g-unif** $\zeta : R[e_j^\sigma] \xrightarrow{\text{ci}} \zeta : R[e^\sigma]$. Thus, by **g-unif**, $\zeta : R[e^\sigma] \downarrow$, and hence $\zeta : R[e_1^\sigma] \downarrow$.

□**Equi-reduct**

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

The proof of this theorem is given in §8, since to establish the result in our general setting we use the uniform computation machinery. In fact we prove

a stronger version that is most conveniently stated using the enriched syntax. The equivalence **(lv.1)** given above is a direct consequence of **(equi-rcx)**.

A global variant of **(equi-reduct)** that applies to any reduction holds if we strengthen the final clause of **g-unif**.

Definition (Strongly Uniform Semantics): A λ -language has *strongly uniform semantics* if it satisfies the properties **g-unif[†]** and **s-unif**, where **g-unif[†]** is the specification obtained by adding the clause **(sred)**

$$\begin{aligned} \text{(sred)} \quad S_0 \longrightarrow S_1 \quad \text{implies} \quad \zeta : R[s2e(S_0)^\sigma] \uparrow \zeta : R[s2e(S_1)^\sigma] \\ \text{for all closing } \zeta, R, \sigma \end{aligned}$$

Although the requirement seems strong, it is in fact easy to show for the languages considered here.

Corollary (redeg): In a λ -language with strongly uniform semantics, reduction preserves operational equivalence: $S_0 \longrightarrow S_1 \Rightarrow s2e(S_0) \cong s2e(S_1)$.

If λ -language has uniform semantics, then reduction steps on states with a free variable p are uniform in the assumption that $p \in \mathbf{L}$. This is made precise in **(L-unif)**.

Theorem (Lambda uniformity (L-unif)): Working in a λ -language with uniform semantics, let $\zeta : e$ be a state with a most one free variable, p . If

$$(\zeta : e)^{\{p \mapsto \lambda x. e_0\}} \longrightarrow \zeta_1 : e_1,$$

then either e has the form $R[\mathbf{app}(p, v)]$ or there is some $\zeta_2 : e_2$ such that

$$(\zeta : e)^{\{p \mapsto \lambda x. e'_0\}} \longrightarrow (\zeta_2 : e_2)^{\{p \mapsto \lambda x. e'_0\}}$$

for any $\lambda x. e'_0 \in \mathbf{L}$. Similarly for multiple free variables assumed to be lambdas.

It is easy to verify **(L-unif)** once the place-holder and uniform computation machinery is in place. **(L-unif)** gives us a simulation-like method for establishing equivalence of lambdas. Intuitively, we can see that one lambda approximates another if any terminating computation containing occurrences of substitution instances of the first can be transformed into a terminating computation containing corresponding occurrences of substitution instances of the second. We use states with variables to mark the corresponding occurrences of lambda instances. Lambda uniformity lets us reduce the work of establishing the correspondence to considering the case when a marked lambda occurrence is applied.

Corollary (L-unif-sim): In a λ -language with uniform semantics, to show that $\varphi_0 \sqsubseteq \varphi_1$, for $\varphi_0, \varphi_1 \in \mathbf{L}$, it suffices to show that for each list of instantiations of the free variables of φ_0, φ_1 , $[\varphi_{j,i} = \varphi_j^{\sigma_i} \mid 0 \leq i \leq n]$, $j < 2$, and each ζ, R, v that closes the instantiated lambdas but may have free variables $\{p_0 \dots p_n\}$ we can find ζ', e' and a list of additional instantiations $[\varphi_{j,i} = \varphi_j^{\sigma_i} \mid n+1 \leq i \leq n+k]$, $j < 2$, such that for any i

1. $\zeta : R[\mathbf{app}(\varphi_{0,i}, v)] \longrightarrow (\zeta' : e')^{\{p_{n+j} \mapsto \varphi_{0,n+j} \mid 1 \leq j \leq k\}}$ in one or more steps, and
2. $(\zeta' : e')^{\{p_i \mapsto \varphi_{1,i} \mid 0 \leq i \leq n+k\}} \preceq (\zeta : R[\mathbf{app}(p_i, v)])^{\{p_i \mapsto \varphi_{1,i} \mid 0 \leq i \leq n\}}$

Proof : Assume the conditions above hold. By **ciu**, to show $\varphi_0 \sqsubseteq \varphi_1$ it suffices to show that

$$(\zeta : e)^{\{p_i \mapsto \varphi_{0,i} \mid 0 \leq i \leq n\}} \preceq (\zeta : e)^{\{p_i \mapsto \varphi_{1,i} \mid 0 \leq i \leq n\}}$$

for each list of instantiations $\varphi_{j,i}$ as above, and each closing $\zeta : e$ with free variables among $\{p_i \mid 0 \leq i \leq n\}$. Assume the left-hand state is defined, then we show by induction on the computation length that the right-hand state is also define. By (**L-unif**) we have three cases to consider.

- (v) $e^{\{p_i \mapsto \varphi_{j,i} \mid 0 \leq i \leq n\}}$ is a value for $j < 2$ and we are done.
- (r) $\zeta : e \longrightarrow \zeta' : e'$ assuming $\{p_i \mid 0 \leq i \leq n\}$ are lambdas. By **g-unif** $\zeta' : e'$ has a smaller computation length, thus by computation induction we are done.
- (p) e has the form $R[\mathbf{app}(p_i, v)]$. By assumption 1. we can find and e' and extend the list of instantiations so that

$$\zeta : R[\mathbf{app}(p_i, v)]^{\{p_i \mapsto \varphi_{0,i} \mid 0 \leq i \leq n\}} \longrightarrow \zeta : e'^{\{p_i \mapsto \varphi_{0,i} \mid 0 \leq i \leq n+k\}}$$

in one or more steps, thus $(\zeta : e'^{\{p_i \mapsto \varphi_{1,i} \mid 0 \leq i \leq n+k\}}) \downarrow$ by computation induction, and by assumption 2. we are done.

□_{L-unif-sim}

To illustrate the use of (**L-unif-sim**) we prove that **Yv** is a least-fixed-point combinator.

Theorem (Least Fix): For F of the form $\lambda f. \lambda x. e \ \mathbf{Yv}(F)$ is the \sqsubseteq -least fixed point of F :

$$\text{(fix)} \quad \mathbf{Yv}(F) \cong F(\mathbf{Yv}(F))$$

$$\text{(least)} \quad F(\varphi) \sqsubseteq \varphi \Rightarrow \mathbf{Yv}(F) \sqsubseteq \varphi$$

Proof : Let $F = \lambda f. \lambda x. e$, $H = \lambda h. \lambda x. F(h(h))(x)$, $\mathbf{Yv}[F] = \lambda x. F(H(H))(x)$, and $F[\mathbf{Yv}[F]] = \lambda x. (e^{\{f \mapsto \mathbf{Yv}[F]\}})$. Here we are using the notation convention that when $\psi(v)$ reduces to a lambda then we write $\psi[v]$ for that lambda. In particular by the rule for **app** we have $\mathbf{Yv}(F) \longrightarrow \mathbf{Yv}[F]$, and $F(\mathbf{Yv}[F]) \longrightarrow F[\mathbf{Yv}[F]]$. With this convention we have $\psi(v) \cong \psi[v]$ by (**Equi-red**). Also, by the rule for **app** we have $\mathbf{Yv}[F](v) \longrightarrow F[\mathbf{Yv}[F]](v)$. To prove (**fix**), using the notation of (**L-unif-sim**), we take $e' = R[e^{\{f \mapsto \mathbf{Yv}[F], x \mapsto v\}}]$ and it is easy to see that the conditions 1,2 hold for both directions of the approximation. To prove (**least**), assume that $F(\varphi) \sqsubseteq \varphi$. Let $e' = R[F(p')(v)]$, with p' fresh. Then $R[\mathbf{Yv}[F](v)] \longrightarrow e'^{\{p' \mapsto \mathbf{Yv}[F]\}}$ and using $F(\varphi)(v) \sqsubseteq \varphi(v)$ we see that conditions 1 and 2 of (**L-unif-sim**) hold and we are done.

3 Functional Primitives

In the example languages considered here, we assume \mathbf{A} contains two distinct atoms playing the role of booleans, \mathbf{t} for *true* and \mathbf{nil} for *false*, and atoms playing the role of the natural numbers, which we denote by $0, 1, \dots$.

The functional language, Λ_f , has operations \mathbf{O}^f where \mathbf{O}^f includes lambda application (\mathbf{app} – arity 2), branching (\mathbf{br} – arity 3), equality on atoms (\mathbf{eq} – arity 2), pairing (\mathbf{pr} , \mathbf{fst} , \mathbf{snd} , \mathbf{ispr} – arities 2,1,1,1), and arithmetic operations ($\mathbf{+1}$, $\mathbf{-1}$, \mathbf{isnat} , \mathbf{iszero} , ... – arities 1,1,1,1, ...). The definition of value expressions of Λ_f is completed by specifying

$$\mathbf{V} = \mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{P}$$

$$\mathbf{P} = \mathbf{pr}(\mathbf{V}, \mathbf{V})$$

The branching primitive is strict, due to the call-by-value semantics. However the usual if-then-else conditional, \mathbf{if} , can be defined as follows.

$$\mathbf{if}(e_0, e_1, e_2) \stackrel{\Delta}{=} \mathbf{app}(\mathbf{br}(e_0, \lambda d.e_1, \lambda d.e_2), \mathbf{nil}) \quad \text{where } d \notin \mathbf{FV}(e_1, e_2)$$

In the functional case, there is no state information and hence only one state context, the initial context, which we represent as the empty context: $\diamond = \bullet$. We write e for $\bullet : e$. Thus, in Λ_f , states are not notationally distinguished from expressions and are self representing – the expression representing a state is $\mathbf{s2e}(e) = \bullet[e] = e$.

We want to specify the rules for the functional primitives in a manner that will work for any λ -language containing these primitives. Because we allow free variables in redexes, to reduce testing operations such as \mathbf{br} , \mathbf{eq} , or \mathbf{ispr} , with a variable argument, it is necessary to know whether or not the variable is bound in the state context, and in complex states more information about the binding may be needed. Thus we define the reduction rules in terms of a satisfaction relation, $\zeta \models \Phi$, between state contexts, ζ , and assertions Φ . This allows us to specialize the reduction rule for an operation to a particular language by completing the specification of the satisfaction relation for state contexts of that language. For the languages considered here, we use assertions about membership and non-membership in sets of values and binary relations on values.

Definition (Assertions): Let W_1 range over sets of values such as $\{\mathbf{nil}\}$, \mathbf{A} , \mathbf{L} , \mathbf{P} , etc. and let W_2 range over binary relations on values. In Λ_f assertions are of one of the following forms: $v \in W_1$; $v \notin W_1$; $(v_0, v_1) \in W_2$; $(v_0, v_1) \notin W_2$.

We first define satisfaction for the empty context (of any language). We then state some further constraints on satisfaction.

Definition (Satisfaction $\diamond \models \Phi$):

$$\mathbf{AtEq} = \{(v, v) \mid v \in \mathbf{A}\}$$

- ◇ $\models v \in W$ iff $v \in W$ for $W \in \{\mathbf{A}, \mathbf{P}, \mathbf{L}, \{\mathbf{nil}\}\}$
- ◇ $\models (v_0, v_1) \in \mathbf{AtEq}$ iff $(v_0, v_1) \in \mathbf{AtEq}$
- ◇ $\models v \notin W$ iff $v \in (\mathbf{A} \cup \mathbf{P} \cup \mathbf{L}) - W$ for $W \in \{\mathbf{A}, \mathbf{P}, \mathbf{L}, \{\mathbf{nil}\}\}$
- ◇ $\models (v_0, v_1) \notin \mathbf{AtEq}$ iff $(v_0, v_1) \in ((\mathbf{A} \cup \mathbf{P} \cup \mathbf{L}) \times (\mathbf{A} \cup \mathbf{P} \cup \mathbf{L})) - \mathbf{AtEq}$

Note that in general a state context provides only partial information and thus need not satisfy either a membership assertion or its corresponding non-membership assertion. For example neither $\diamond \models x \in \mathbf{A}$ nor $\diamond \models x \notin \mathbf{A}$ holds for a variable x . We require that if an assertion holds in the empty context, then it holds in any context. Also satisfaction must be preserved by value substitution.

Specification (Satisfaction $\zeta \models \Phi$):

- ◇ $\models \Phi \Rightarrow \zeta \models \Phi$
- $\zeta \models \Phi \Rightarrow \zeta^\sigma \models \Phi^\sigma$ if $\text{Dom}(\sigma) \cap \text{Traps}(\zeta) = \emptyset$

For static assertions such as those defined above, it will be the case that $\zeta \models \Phi$ and $\zeta : e \longrightarrow \zeta' : e'$ implies $\zeta' \models \Phi$, but we don't make this an official requirement.

Since neither the state context nor the reduction context is changed by reducing a functional redex, we define a local reduction relation $r \hookrightarrow_\zeta e$, then lift this to states in the obvious manner. As examples of local functional reduction rules we give the rule for application (aka beta-v) rule, and the rules for branching, projection, and the tests for pairs, and equality.

Definition (Functional rules):

- (app) $\text{app}(\lambda x.e, v) \hookrightarrow_\zeta e^{\{x \mapsto v\}}$
- (br) $\text{br}(v_0, v_1, v_2) \hookrightarrow_\zeta \begin{cases} v_1 & \text{if } \zeta \models v \notin \{\mathbf{nil}\} \\ v_2 & \text{if } \zeta \models v \in \{\mathbf{nil}\} \end{cases}$
- (fst) $\text{fst}(\text{pr}(v_0, v_1)) \hookrightarrow_\zeta v_0$
- (ispr) $\text{ispr}(v) \hookrightarrow_\zeta \begin{cases} \mathbf{t} & \text{if } \zeta \models v \in \mathbf{P} \\ \mathbf{nil} & \text{if } \zeta \models v \notin \mathbf{P} \end{cases}$
- (eq) $\text{eq}(v_0, v_1) \hookrightarrow_\zeta \begin{cases} \mathbf{t} & \text{if } \zeta \models (v_0, v_1) \in \mathbf{AtEq} \\ \mathbf{nil} & \text{if } \zeta \models (v_0, v_1) \notin \mathbf{AtEq} \end{cases}$
- (rdx) $\zeta : R[r] \longrightarrow \zeta : R[e]$ if $r \hookrightarrow_\zeta e$

It is easy to see from the form of the rules that **g-unif** holds in Λ_f . In addition we have the following properties of computation in Λ_f .

Lemma (fred): In Λ_f

- (R-unif) $R[e_0] \longrightarrow R[e_1] \Rightarrow R'[e_0] \longrightarrow R'[e_1]$ for any $R, R', e_0 \notin \mathbf{V}$
- (isdef) $R[e] \downarrow \Rightarrow (\exists v') R[e] \longrightarrow R[v']$

The following is a sampling of the equational laws for the functional operations. The first two laws correspond to the laws of the computational lambda calculus (Moggi 1988).

Theorem (Functional laws): In any λ -language with uniform semantics extending Λ_f we have

$$\begin{aligned}
(\text{betav}) \quad & \text{app}(\lambda x.e, v) \cong e^{\{x \mapsto v\}} \\
(\text{dist}) \quad & (\text{let } x = e \text{ in } R[x]) \cong R[e] \quad \text{if } x \notin \text{FV}(R) \\
(\text{if.dist}) \quad & R[\text{if}(e, e_1, e_2)] \cong \text{if}(e, R[e_1], R[e_2]) \\
(\text{proj}) \quad & \text{fst}(\text{pr}(x, y)) \cong x \quad \text{and} \quad \text{snd}(\text{pr}(x, y)) \cong y \\
(\text{if}) \quad & \text{if}(\text{t}, e_1, e_2) \cong e_1 \quad \text{and} \quad \text{if}(\text{nil}, e_1, e_2) \cong e_2
\end{aligned}$$

Proof : (**betav,if,proj**) follow from (**equi-red**) since the equated expressions have a common reduct. (**dist,if.dist**) follow from (**equi-rcx**) (the stronger form is needed for (**if.dist**)) since the equated expressions are the result of placing a common expression, e in equivalent reduction contexts.

□**Functional laws**

Since functional computation has no effects, two computations that do not use each other's results can be permuted.

Lemma (perm): In Λ_f , if $x \notin \text{FV}(e_1)$ and $y \notin \text{FV}(e_0)$, then

$$(\text{let } x = e_0 \text{ in let } y = e_1 \text{ in } e) \cong (\text{let } y = e_1 \text{ in let } x = e_0 \text{ in } e)$$

This law relies strongly on the (**fred.isdef**) property, and fails in various ways, as we shall see, in the presence of effects.

3.1 Programming Examples

Everywhere Undefined Functions

By (**ciu**) it is easy to see that any two undefined expressions are equivalent. Abstracting and using (**L-unif**) we have that any two lambdas that are everywhere undefined are equivalent. The classic example of an everywhere undefined lambda is

$$\text{Bot} \triangleq \lambda x.\text{app}(\lambda x.\text{app}(x, x), \lambda x.\text{app}(x, x))$$

In Λ_f , another example of an everywhere undefined lambda is the “do-forever” loop.

$$\text{Do} \triangleq \lambda f.\text{Yv}(\lambda \text{Do} \lambda x.\text{Do}(f(x)))$$

By the recursive definition, for any lambda φ and value v

$$\text{Do}(\varphi)(v) \longrightarrow \text{Do}(\varphi)(\varphi(v))$$

In Λ_f , either $\varphi(v) \longrightarrow v'$ for some v' or $\varphi(v)$ is undefined. In the latter case the computation is undefined since the redex is undefined. In the former case, the computation reduces to $\text{Do}(\varphi)(v')$ and on we go. The argument for undefinedness of **Bot** relies only on the (**app**) rule and will be valid in any uniform semantics. In contrast the argument for undefinedness of $\text{Do}(\varphi)$ relies on the (**fred.isdef**) property of Λ_f .

Functional Streams

We now illustrate the use of (**L-unif-sim**) computation to reason about streams represented as functions which when accessed (applied) return a pair consisting of the next stream element and the function representing the remainder of the stream. As Gordon (1995) and others have shown, bisimulation methods are also well suited to reasoning about equivalence of functional streams. We will see later that (**L-unif-sim**) generalizes nicely for reasoning about objects with private memory. $\text{NumS}(k)$ is the stream of numbers in increasing order beginning with k , and $\text{OddS}(k)$ is the stream of odd numbers starting with $2k + 1$. $\text{Alt}(s)$ is the stream obtained by removing every other element of s .

$$\begin{aligned} \text{NumS} &\triangleq \text{Yv}(\lambda \text{NumS}.\lambda k.\lambda d.\text{pr}(k, \text{NumS}(k + 1))) \\ \text{Alt} &\triangleq \text{Yv}(\lambda \text{Alt}.\lambda s.\lambda d.\text{let } xs = s(\text{nil}) \text{ in} \\ &\quad \text{let } ys = \text{snd}(xs)(\text{nil}) \text{ in} \\ &\quad \text{pr}(\text{fst}(xs), \text{Alt}(\text{snd}(ys)))) \\ \text{OddS} &\triangleq \text{Yv}(\lambda \text{OddS}.\lambda k.\lambda d.\text{pr}(2k + 1, \text{OddS}(k + 1))) \end{aligned}$$

Lemma (odds): $\text{OddS}(k) \cong \text{Alt}(\text{NumS}(2k + 1))$.

Proof : Let $\varphi_{0,k} = \text{OddS}(k)$ and $\varphi_{1,k} = \text{Alt}(\text{NumS}(2k + 1))$. By the computation rules for **OddS**, **Alt**, **NumS**, for any $\zeta, R, k \in \mathbf{N}$,

$$\zeta : R[\varphi_{j,k}(v)] \longrightarrow (\zeta : R[\text{pr}(2k + 1, p_{k+1})])^{\{p_{k+1} \mapsto \varphi_{j,k+1}\}}$$

and it is easy to see that the (**L-unif-sim**) conditions hold in both directions of approximation.

□_{odds}

4 Control Effects

Now we introduce control effects, adding a new primitive operator **ncc** (for Note Current Continuation). The language with control facilities, Λ_c , has operations $\mathbf{O}^c = \mathbf{O}^f \cup \{\text{ncc}\}$.

Roughly speaking, $\text{ncc}(v)$ captures the current reduction context as a continuation, and applies v to this continuation at the top level. ncc is called \mathcal{C} in (Felleisen and Friedman 1986; Felleisen 1987). ncc differs from the Scheme call/cc primitive (Steele and Sussman 1975; Rees and Clinger 1986) in that call/cc evaluates the application of v to the captured continuation in the context of that continuation rather than discarding it. call/cc can be defined using ncc as follows.

$$\text{call/cc} \triangleq \lambda f.\text{ncc}(\lambda c.c(f(c)))$$

ncc can also be used to define an abort primitive. We call this top and we use it to represent the top level of a computation. top , simply returns its argument to the top level.

$$\text{top} \triangleq \lambda x.\text{ncc}(\lambda k.x)$$

As in the functional case, Λ_c has only one state context, the initial context, $\diamond_c = \text{top}(\bullet)$. Thus, a Λ_c state has the form $\diamond_c : e$ and the expression associated to a Λ_c state is defined by $\text{s2e}(\diamond_c : e) = \text{top}(e)$. Even though there is only one state context, we keep it explicit in our notation to emphasize the distinction between top level in the presence of control primitives and the simple functional case. Reduction rules for functional operations apply directly to Λ_c states since satisfaction for \diamond is the same in any language. Thus, we need only supply a reduction rule for ncc .

Definition (Ncc reduction):

$$(\text{ncc}) \quad \zeta : R[\text{ncc}(v)] \longrightarrow \zeta : \text{app}(v, \text{top} \circ R)$$

We use the convention that R used where a lambda should appear abbreviates $\lambda x.R[x]$ for some $x \notin \text{FV}(R)$. As for the functional primitives, we have specified the rule for ncc in a manner that defines the rule for any λ -language containing the ncc operation. We say a λ -language has *ncc control*, if ncc is among the operations of that λ -language, the reduction rule for ncc in that λ -language is that given above, and no other reduction rules manipulate the reduction context. In particular rules for primitives other than ncc will have the form $\zeta : R[\vartheta(v_1, \dots, v_n)] \longrightarrow \zeta' : R[e]$. We introduce the notion of λ -language with *ncc control* to characterize a class of languages for which the basic ncc laws hold. As we will see below, permitting other control primitives can invalidate these laws.

Lemma (top rule): The derived reduction rule for top is

$$(\text{top}) \quad \zeta : R[\text{top}(v)] \longrightarrow \zeta : v$$

Proof : Using the Λ_c computation rules we have

$$\zeta : R[\text{top}(v)] \triangleq \zeta : R[(\lambda x.\text{ncc}(\lambda c.x))(v)]$$

$$\begin{aligned}
&\longrightarrow \zeta : R[\text{ncc}(\lambda c.v)] \quad c \notin \text{FV}(v) \\
&\longrightarrow \zeta : \text{app}(\lambda c.v, \text{top} \circ R) \\
&\longrightarrow \zeta : v
\end{aligned}$$

□_{top rule}

To provide further intuition about computation with `ncc`, we introduce a useful abbreviation, `note(c)e`, and derive its computation rule. `note(c)e` binds c in e to the current continuation and arranges for e to be evaluated without discarding the current continuation. `note` is to `call/cc` what `let` is to λ .

$$\text{note}(c)e \triangleq \text{ncc}(\lambda c.\text{app}(c, e))$$

Lemma (note rule):

$$(\text{note}) \quad \zeta : R[\text{note}(c)e] \longrightarrow \zeta : \text{app}(\text{top} \circ R, e^{\{c \mapsto \text{top} \circ R\}})$$

Proof :

$$\begin{aligned}
&\zeta : R[\text{note}(c)e] \triangleq \zeta : R[\text{ncc}(\lambda c.c(e))] \\
&\longrightarrow \zeta : \text{app}(\lambda c.c(e), \text{top} \circ R) \\
&\longrightarrow \zeta : \text{app}(\text{top} \circ R, e^{\{c \mapsto \text{top} \circ R\}})
\end{aligned}$$

□_{note rule}

It is again easy to see that **g-unif** holds in Λ_c . However, the functional reduction properties (**fred**) fail. To see this, note that $\zeta : R[\text{top}(0)] \downarrow$ but for non-empty R , there is no value v such that $\zeta : R[\text{top}(0)]$ reduces to $\zeta : R[v]$.

A lemma that is useful in dealing with the top-level is the following.

Lemma (top.elim): In a λ -language with uniform semantics and `ncc` control

- (1) $\text{top} \cong R \circ \text{top}$
- (2) $\zeta : R[\text{top}(e)] \uparrow \zeta : e$ for ζ closed R, e

Proof : To show (1) we use (**L-unif**) noting that by the `top` rule, $\zeta : R'[\text{app}(\varphi, v)] \longrightarrow \zeta : v$ for $\varphi \in \{\text{top}, R \circ \text{top}\}$. For (2) if $e \in \mathbf{V}$ we are done (by the `top` rule). Also if the computation leading from $\zeta : e$ does not invoke the `ncc` rule, then (by the `ncc` control assumption) either both states are undefined, or both lead to the same value state. Otherwise suppose $\zeta : e \longrightarrow \zeta' : R[\text{ncc}(v)]$ by steps not involving `ncc`. Then $\zeta : e \longrightarrow \zeta' : \text{app}(v, \text{top} \circ R)$, and $\zeta : \text{top}(e) \longrightarrow \zeta' : \text{app}(v, \text{top} \circ \text{top} \circ R)$. By (1) and the functional laws, $\text{top} \cong \text{top} \circ \text{top}$ and we are done. **□_{top.elim}**

(**top.elim**) is used in establishing a number of basic **ncc** laws. To see how this can fail in the presence of other control primitives, consider adding **fcc** defined by the rule $\zeta : R[\mathbf{fcc}(v)] \longrightarrow \zeta : \mathbf{app}(v, R)$. The continuation captured by **fcc** composes with rather than escaping from any surrounding context when it is applied. Let $e = \mathbf{fcc}(\lambda k. \mathbf{let } x = k(v) \mathbf{ in } \mathbf{Bot})$. Then $\zeta : \mathbf{top}(e) \longrightarrow \zeta : v$ while $\zeta : e \longrightarrow \zeta : \mathbf{Bot}$ and hence is undefined.

The following is a sampling of the laws axiomatizing **ncc**.

Theorem (Ncc laws (ncc)): In any λ -language with uniform semantics and **ncc** control

- (1) $\mathbf{ncc}(\lambda c. \mathbf{ncc}(e)) \cong \mathbf{ncc}(\lambda c. \mathbf{app}(e, \mathbf{top}))$
- (2) $R[\mathbf{ncc}(e)] \cong \mathbf{ncc}(\lambda c. \mathbf{app}(e, c \circ R)) \quad c \notin \mathbf{FV}(e, R)$
 $\mathbf{ncc}(f) \cong \mathbf{ncc}(\mathbf{top} \circ f)$
- (3) $\mathbf{ncc}(\lambda c. C[c]) \cong \mathbf{ncc}(\lambda c. C[\mathbf{top} \circ c])$
- (4) $\mathbf{note}(c)e \cong e \quad \text{if } c \notin \mathbf{FV}(e)$
- (5) $\mathbf{note}(c)R[e] \cong \mathbf{let } x = e \mathbf{ in } \mathbf{note}(c)R[x] \quad \text{if } c \notin \mathbf{FV}(e), x \notin \mathbf{FV}(R)$
- (6) $\mathbf{note}(c)\mathbf{if}(e_0, e_1, e_2) \cong \mathbf{if}(e_0, \mathbf{note}(c)e_1, \mathbf{note}(c)e_2) \quad \text{if } c \notin \mathbf{FV}(e_0)$

Proof : The general idea for establishing (**ncc**) is the following. To show $e_0 \cong e_1$, by **ciu** it suffices to show that

$$\zeta : R^*[e_0^\sigma] \Downarrow \zeta : R^*[e_1^\sigma]$$

for any closing ζ, R^*, σ . To do this we proceed as follows (except for (3) which follows by direct calculation).

1. Find R_0, R_1, e_{01} such that $\zeta : R^*[e_j^\sigma] \longrightarrow \zeta : R_j[e_{01}]$ for $j < 2$
2. Show that $\mathbf{top} \circ R_0 \cong \mathbf{top} \circ R_1$

Then for $j < 2$

$$\zeta : R^*[e_j^\sigma] \Downarrow \zeta : R_j[e_{01}] \quad \text{by 1.}$$

$$\zeta : R_j[e_{01}] \Downarrow \zeta : \mathbf{top}(R_j[e_{01}]) \quad \text{by } (\mathbf{top.elim})$$

$$\zeta : \mathbf{top}(R_j[e_{01}]) \Downarrow \zeta : (\mathbf{top} \circ R_j)(e_{01}) \quad \text{by } (\mathbf{dist})$$

and by condition 2. we are done.

As an example we carry out this process for (**ncc.5**). In this case we have $e_0 = \mathbf{note}(c)R[e]$ and $e_1 = \mathbf{let } x = e \mathbf{ in } \mathbf{note}(c)R[x]$ where $c \notin \mathbf{FV}(e)$ and $x \notin \mathbf{FV}(R)$. Pick some closing ζ, R^*, σ , and let

$$R_0 = (\mathbf{top} \circ R^*)(R^{\{c \mapsto \mathbf{top} \circ R^*\}})$$

$$R_1 = R^*[\mathbf{let } x = \bullet \mathbf{ in } \mathbf{note}(c)R[x]]$$

$$e_{01} = e$$

Then using the `note` rule and (**L-unif**) it is easy to check that conditions 1. and 2. hold.

□ **Ncc laws**

A somewhat non-intuitive equivalence that holds in λ -languages with `ncc` control is $\text{top}(v) \cong \text{top}(v')$ for any v and v' . This simply expresses the fact that values returned by `top` can not be observed by any program. A refined notion of equivalence for which this equation holds only when the value expressions are equivalent, but for which the (**Ncc laws**) are valid was studied in (Talcott 1989).

Note that $\text{Do}(f) \cong \text{Bot}$ fails in Λ_c . The reason is the ability to escape from a loop. A distinguishing context is `note(c) let f = $\lambda x.$ if(iszero(x), c(0), x - 1) in app(\bullet , 0)`.

(**perm**) also fails in Λ_c , since permutation of the order of expression evaluation can change termination properties. As a counterexample let $e_0 = \text{ncc}(\lambda k.0)$ and $e_1 = \text{Bot}(0)$, then

$$\begin{aligned} \zeta : \text{let } x_0 &= \text{ncc}(\lambda k.0) \text{ in let } x_1 = \text{Bot}(0) \text{ in } e \longrightarrow \zeta : 0 \\ \zeta : \text{let } x_1 &= \text{Bot}(0) \text{ in let } x_0 = \text{ncc}(\lambda k.0) \text{ in } e \\ &\longrightarrow \zeta : \text{let } x_1 = \text{app}(\lambda x.x(x), \lambda x.x(x)) \text{ in let } x_0 = \text{ncc}(\lambda k.0) \text{ in } e \\ &\longrightarrow \dots \quad \text{forever} \end{aligned}$$

The following lemma shows that `call/cc` and `note` are inter-definable, and `ncc` is definable from `call/cc` and `top`.

Lemma (control):

- (1) $\text{call/cc} \cong \lambda f.\text{note}(c)f(c)$
- (2) $\text{note}(c)e \cong \text{call/cc}(\lambda c.e)$
- (3) $\text{ncc} \cong \lambda f.\text{call/cc}(\lambda c.\text{top}(f(c)))$

Proof : (1,2) follow by expanding the definitions, possibly using (**betav**). (3) requires an application of (**ncc.2**) as well. □

5 Memory Effects

Now we consider memory effects. The language Λ_m has operations $\mathbf{O}^m = \{\text{mk}, \text{get}, \text{set}, \text{iscell}\} \cup \mathbf{O}^f$. Intuitively, `mk`(v) allocates a new cell containing v and returns that cell, `get`(z) returns the contents of the cell z , `set`(z, v) sets the contents of the cell z to be v , and `iscell`(v) tests whether v is a cell.

A state context of Λ_m is a memory context, M , of the form

$$M = \text{let } z_1 = \text{mk}(\text{nil}) \text{ in}$$

$$\dots$$

$$\text{let } z_k = \text{mk}(\text{nil}) \text{ in}$$

$$\text{set}(z_1, v_1); \dots \text{set}(z_k, v_k); \bullet$$

The **lets** of M 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 **let** $z = \text{mk}(\text{nil})$ **in** **set**(z, z); \bullet . This is not the same as the context **let** $z = \text{mk}(z)$ **in** \bullet , since in the latter case the z in the argument to **mk** is bound outside the context and is distinct from the created z . 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. For M as above, $\text{Dom}(M) = \{z_1, \dots, z_k\}$, $M(z_i) = v_i$ for $1 \leq i \leq k$, and we write $\{z_i \mapsto \text{mk}(v_i) \mid 1 \leq i \leq k\}$ for M . 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 M the ambiguity makes no difference. The empty state context of Λ_m is the empty context, \bullet , and the map associating Λ_m states to expressions is defined by $\text{s2e}(M : e) = M[e]$.

To define the reduction relation for the new Λ_m operations two new assertions – $v \in \text{Cell}$ and $v \notin \text{Cell}$ – are needed. The definition of satisfaction for Λ_m state contexts is completed as follows.

Definition (Satisfaction for memory contexts):

$$M \models v \in \text{Cell} \quad \text{iff} \quad v \in \text{Dom}(M)$$

$$M \models v \notin \text{Cell} \quad \text{iff} \quad v \notin \text{Dom}(M)$$

$$M \models v \notin W \quad \text{if} \quad v \in \text{Dom}(M) \quad \text{for} \quad W \in \{\mathbf{A}, \mathbf{L}, \mathbf{P}, \{\text{nil}\}\}$$

Definition (Memory Rules):

$$(\text{iscell}) \quad \text{iscell}(v) \hookrightarrow_M \begin{cases} \mathbf{t} & \text{if } M \models v \in \text{Cell} \\ \text{nil} & \text{if } M \models v \notin \text{Cell} \end{cases}$$

$$(\text{mk}) \quad M : R[\text{mk}(v)] \longrightarrow M\{z \mapsto \text{mk}(v)\} : R[z]$$

$$z \notin (\text{Dom}(M) \cup \text{FV}(M[R[v]]))$$

$$(\text{get}) \quad M : R[\text{get}(z)] \longrightarrow M : R[v] \quad \text{if} \quad M(z) = v$$

$$(\text{set}) \quad M : R[\text{set}(z, v)] \longrightarrow M\{z \mapsto \text{mk}(v)\} : R[\text{nil}] \quad \text{if} \quad z \in \text{Dom}(M)$$

Recall from §3. that local reduction lifts according to the (**rdx**) rule: $M : R[r] \longrightarrow M : R[e]$ if $r \hookrightarrow_M e$. Note that the rule for **get** could also have been expressed as a local rule: $\mathbf{get}(v) \hookrightarrow_M v'$ if $v \in \text{Dom}(M)$ and $Mv = v'$. On the other hand, the rules for **mk** and **set** can not be formulated as local rules.

It is again easy to verify that **g-unif** holds in Λ_m . The unicity property makes explicit the fact that, in our model, arbitrary choice in cell allocation is the same phenomenon as arbitrary choice of names of bound variables. The following analog of (**fred**) holds in Λ_m .

Lemma (mred): In Λ_m

$$(\text{R-unif}) \quad M : R[e] \longrightarrow M' : R[e'] \Rightarrow M : R'[e] \longrightarrow M' : R'[e']$$

$$\text{if } e \notin \mathbf{V}, (\text{Dom}(M') \cap \text{FV}(R')) \subseteq \text{Dom}(M)$$

$$(\text{isdef}) \quad M : R[e] \downarrow \Rightarrow (\exists M', v)(M : R[e] \longrightarrow M' : R[v])$$

Some further simple consequences of the computation rules are that memory contexts may be pulled out of reduction contexts, and that computation is uniform in unreferenced memory.

Lemma (umem): In Λ_m

$$(1) \quad \bullet : R[M[e]] \longrightarrow M : R[e] \quad \text{if } \text{FV}(R) \cap \text{Dom}(M) = \emptyset.$$

$$(2) \quad M : e \longrightarrow M' : e' \Rightarrow (M_0 \cup M) : e \longrightarrow (M_0 \cup M') : e' \\ \text{if } \text{Dom}(M_0) \cap \text{Dom}(M') = \emptyset$$

$$(2a) \quad (M_0 \cup M) : e \longrightarrow (M_0 \cup M') : e' \Rightarrow M : e \longrightarrow M' : e' \\ \text{if } (\text{Dom}(M') \cup \text{FV}(M[e])) \cap \text{Dom}(M_0) = \emptyset$$

Note that in (**umem.2**) the if clause implies that $\text{Dom}(M) \cap \text{Dom}(M_0) = \emptyset$.

An example computation

Num0 is the mutable analog of **NumS**. The value of **Num0**(k) is the ‘object’ with script **Num0a**(z) where z is a private cell (accessible only from within **Num0a**(z)) whose initial contents is k . When queried, **Num0a**(z) returns the contents of z and increments that contents by 1, thus generating the stream of numbers.

$$\mathbf{Num0} \triangleq \lambda x. \mathbf{Num0a}(\mathbf{mk}(x))$$

$$\mathbf{Num0a} \triangleq \lambda z. \lambda d. \mathbf{let } x = \mathbf{get}(z) \mathbf{ in set}(z, x + 1); x$$

The computation rules for **Num0** and **Num0a** are given by the following lemma.

Lemma (Num0 rules): For $z \notin (\text{Dom}(M) \cup \text{FV}(R))$, $k \in \mathbf{N}$

$$(\text{numo}) \quad M : R[\mathbf{Num0}(k)] \longrightarrow M\{z \mapsto \mathbf{mk}(k)\} : R[\mathbf{Num0a}(z)]$$

$$(\text{numa}) \quad M\{z \mapsto \mathbf{mk}(k)\} : R[\mathbf{Num0a}(z)(v)] \longrightarrow M\{z \mapsto \mathbf{mk}(k + 1)\} : R[k]$$

Proof :

$$\begin{aligned}
\bullet : \text{NumO}(k) &= \bullet : \lambda x. \text{NumOa}(\text{mk}(x))(k) && \text{by definition} \\
&\longrightarrow \bullet : \text{NumOa}(\text{mk}(k)) && (\text{app}) \\
&\longrightarrow \{z \mapsto \text{mk}(k)\} : \text{NumOa}(z) && (\text{mk,app})
\end{aligned}$$

and

$$\begin{aligned}
&\{z \mapsto \text{mk}(k)\} : \text{NumOa}(z)(v) \\
&\longrightarrow \{z \mapsto \text{mk}(k)\} : \text{let } x = \text{get}(z) \text{ in set}(z, x + 1); x && (\text{app}) \\
&\longrightarrow \{z \mapsto \text{mk}(k)\} : \text{set}(z, k + 1); k && (\text{get,app}) \\
&\longrightarrow \{z \mapsto \text{mk}(k + 1)\} : k && (\text{set,app})
\end{aligned}$$

Thus using (**mred.R-unif,umem.2**) we are done.

□**NumOrules**

The following equivalences are a sampling of the laws axiomatizing memory operations. Here **eqc** extends the definition of **eq** to reference cells. This can be taken as primitive, or defined as shown in Mason (1986). The (derived) computation rule for **eqc** is given by

$$\begin{aligned}
\mathbf{CEq}[M] &= \{(v, v) \mid v \in \text{Dom}(M)\} \cup \mathbf{AtEq} \\
M \models (v_0, v_1) \in \mathbf{CEq} &\text{ iff } (v_0, v_1) \in \mathbf{CEq}[M] \\
\diamond \models (v_0, v_1) \notin \mathbf{CEq} &\text{ iff } (v_0, v_1) \in (\mathbf{A} \cup \mathbf{P} \cup \mathbf{L} \cup \text{Dom}(M))^2 - \mathbf{CEq}[M] \\
\text{eqc}(v_0, v_1) \hookrightarrow_M &\begin{cases} \mathbf{t} & \text{if } M \models (v_0, v_1) \in \mathbf{CEq} \\ \mathbf{nil} & \text{if } M \models (v_0, v_1) \notin \mathbf{CEq} \end{cases}
\end{aligned}$$

Theorem (Memory laws (mem)): In Λ_m

- (1) $\text{get}(\text{mk}(x)) \cong x$
- (2) $\text{let } x = \text{mk}(v) \text{ in } R[\text{eqc}(x, y)] \cong \text{let } x = \text{mk}(v) \text{ in } R[\mathbf{nil}]$
- (3) $\text{let } x = \text{mk}(v) \text{ in set}(x, v'); e \cong \text{let } x = \text{mk}(v') \text{ in } e$ if $x \notin \text{FV}(v')$
- (4) $\text{set}(x, v); \text{get}(x) \cong \text{set}(x, v); v$
- (5) $\text{set}(x, v); \text{set}(y, v') \cong \text{if}(\text{eqc}(x, y), \text{set}(x, v'), \text{set}(y, v'); \text{set}(x, v))$
- (6) $M[e] \cong e$ if $\text{FV}(e) \cap \text{Dom}(M) = \emptyset$

Proof : The memory laws are all established by the following general argument, which formalizes the intuition that two expressions are equivalent if they reduce to the same expression, with the same effects on memory ignoring inaccessible memory (garbage). To show that $e_0 \cong e_1$ by (**ciu**) pick an arbitrary closing M^*, R^*, σ and show that $M^* : R^*[e_0^\sigma]$ and $M^* : R^*[e_1^\sigma]$

are equidefined. To do this, we show that there are e' (the common reduct), M' (the result of the common effects on M), and M_j (the garbage) such that $\text{Dom}(M_j) \cap (\text{Dom}(M') \cup \text{FV}(M'[e'])) = \emptyset$ and

$$M^* : e_j^{\sigma^*} \longrightarrow M_j \cup M' : e'$$

Then by (**mred.R-unif**)

$$M^* : R^*[e_j^\sigma] \longrightarrow M_j \cup M' : R[e']$$

and by (**umem.2**) the two states $M_j \cup M' : R[e']$ are equidefined. Thus we are done. As an example of this argument, consider memory law (1), with M^*, R^*, σ as above. Thus $e_0 = \text{get}(\text{mk}(x))$ and $e_1 = x$. We let $M_0 = \{z \mapsto \text{mk}(\sigma(x))\}$, $M_1 = \bullet$, $M' = M^*$, and $e' = \sigma(x)$.

□**MemoryLaws**

To simplify the presentation, we have given the rules and equational theory for the memory operations only for Λ_m . This can be generalized to a wide class of λ -languages by identifying conditions that prevent interference with the memory operations as was done for the theory of **ncc**.

The (**perm**) law fails in Λ_m . A simple counterexample is obtained by taking $e_0 = \text{set}(z, 0)$, $e_1 = \text{set}(z, 1)$, and $e = \text{get}(z)$. Then by (**mem.4,5**)

$$\text{let } x = e_0 \text{ in let } y = e_1 \text{ in } e \cong \text{let } y = e_1 \text{ in } 1$$

$$\text{let } y = e_1 \text{ in let } x = e_0 \text{ in } e \cong \text{let } x = e_0 \text{ in } 0$$

However, allocation of memory can be permuted with evaluation of expressions that have no access to that memory. This is a key law for reasoning about programs that manipulate memory (see Mason and Talcott (1990, 1992, 1994a) for some examples).

Lemma (delay): In Λ_m , if $x \notin \text{FV}(e_1)$ and $y \notin \text{FV}(v)$, then

$$(\text{let } x = \text{mk}(v) \text{ in let } y = e_1 \text{ in } e) \cong (\text{let } y = e_1 \text{ in let } x = \text{mk}(v) \text{ in } e)$$

In Λ_m , (**delay**) is a fairly easy consequence of (**ciu**) and (**mred.isdef**). A proof appears in (Mason and Talcott 1991a). As an example of the use of the memory and delay laws we show that $\text{eqc}(\text{mk}(0), \text{mk}(0)) \cong \text{nil}$

$$\begin{aligned} & \text{eqc}(\text{mk}(0), \text{mk}(0)) \\ & \cong \text{let } x = \text{mk}(0) \text{ in let } y = \text{mk}(0) \text{ in eqc}(x, y) \quad (\mathbf{dist}) \text{ twice} \\ & \cong \text{let } y = \text{mk}(0) \text{ in let } x = \text{mk}(0) \text{ in eqc}(x, y) \quad (\mathbf{delay}) \\ & \cong \text{let } y = \text{mk}(0) \text{ in let } x = \text{mk}(0) \text{ in nil} \quad (\mathbf{mem.2}) \\ & \cong \text{nil} \quad (\mathbf{mem.6}) \end{aligned}$$

Stream Object equivalence

As a further example of using uniform computation techniques we show the equivalence of two stream objects built using the mutable analogs `NumO`, `AltO`, and `OddO` of `NumS`, `Alt`, and `OddS`. `NumO` was defined above.

$$\text{OddO} \triangleq \lambda k. \text{OddOa}(\text{mk}(k))$$

$$\text{OddOa} \triangleq \lambda z. \lambda d. \text{let } x = \text{get}(z) \text{ in set}(z, x + 1); 2x + 1$$

$$\text{AltO} \triangleq \lambda s. \lambda d. \text{let } x = \text{app}(s, \text{nil}) \text{ in app}(s, \text{nil}); x$$

Lemma (mutable stream): $\text{OddO}(k) \cong \text{AltO}(\text{NumO}(2k + 1))$

Proof : The proof indicates how to generalize (**L-unif-sim**) to objects. Since objects have private memory, this has to be done with a bit of care. By **ciu** we need only show that

$$M : R[\text{OddO}(k)] \Downarrow M : R[\text{AltO}(\text{NumO}(2k + 1))].$$

By **g-unif** and the derived computation rules for the defined objects, we need only show that

$$M\{z \mapsto \text{mk}(k)\} : R[\text{OddOa}(z)] \Downarrow M\{z \mapsto \text{mk}(2k + 1)\} : R[\text{AltO}(\text{NumOa}(z))]$$

for $k \in \mathbf{N}$. We do this by defining a notion of similar states and showing by computation induction that similar states are equi-defined. Similar states are states of the form

$$S_o = (M\{z \mapsto \text{mk}(k)\} : e)^{\{o \mapsto \text{OddOa}(z)\}}$$

and

$$S_a = (M\{z \mapsto \text{mk}(2k + 1)\} : e)^{\{o \mapsto \text{AltO}(\text{NumOa}(z))\}}$$

for $M : e$ such that $z \notin \text{Dom}(M) \cup \text{FV}(e)$ and $\text{FV}(M[e]) = \{o\}$. To show equi-definedness, assume $S_o \Downarrow$ or $S_a \Downarrow$. If $e \in \mathbf{V}$ then we are done. Otherwise, by (**L-unif**) either $M : e \longrightarrow M' : e'$ assuming $o \in \mathbf{L}$ (and we are done since we have smaller computations of the same form), or e has the form $R[\text{app}(o, v)]$. In this case

$$S_o \longrightarrow (M\{z \mapsto \text{mk}(k + 1)\} : R[2k + 1])^{\{o \mapsto \text{OddOa}(z)\}}$$

and

$$S_a \longrightarrow (M\{z \mapsto \text{mk}(2k + 3)\} : R[2k + 1])^{\{o \mapsto \text{AltO}(\text{NumOa}(z))\}}$$

and again we have the desired smaller computations of the same form.

□mutablestream

More examples of proof principles (and proofs) for reasoning about mutable streams and other forms of object, including a simulation induction principle that abstracts and generalizes the above argument, can be found in Mason and Talcott (1994a, 1994b).

6 Control + Memory Effects

We combine the control and memory operations to obtain a Scheme-like language, Λ_s . The operations of Λ_s , $\mathbf{O}^s = \mathbf{O}^c \cup \mathbf{O}^m$, are the union of the operations of Λ_m and Λ_c . State contexts of Λ_s combine the memory context of Λ_m and the top marker of Λ_c , and are of the form $M[\mathbf{top}(\bullet)]$. We write $M_T : e$ for the Λ_s state with expression e and state context $M[\mathbf{top}(\bullet)]$. As usual, the expression associated to such a state is defined by $\mathbf{s2e}(M_T : e) = M[\mathbf{top}(e)]$.

The reduction relation in Λ_s is obtained by combining the rules for operations of Λ_m and Λ_c . This works, because adding the top marker does not change satisfaction, and the operation \mathbf{ncc} is uniform in the state context.

Definition (Λ_s reduction rules):

$$\begin{aligned} M_T : e &\longrightarrow M_T : e' && \text{if } \diamond_c : e \longrightarrow \diamond_c : e' \\ M_T : e &\longrightarrow M'_T : e' && \text{if } M : e \longrightarrow M' : e' \end{aligned}$$

Note that $M[\mathbf{top}(e)] \cong \mathbf{top}(M[e])$, thus we could have used state contexts of the form $\mathbf{top}(M)$ without changing the induced approximation and equivalence relations.

The properties of memory computation (**umem.1,2,2a**) persist, and the semantics of Λ_s is uniform. Thus, the functional, \mathbf{ncc} , and memory laws hold in Λ_s . Also, the (**top.elim**) property of Λ_c holds in Λ_s , but (**mred**) fail in Λ_s for the same reasons that (**fred**) fails in Λ_c .

As pointed out by Felleisen (1993), (**delay**) fails in Λ_s if the expression whose evaluation is permuted with memory allocation has control effects. An example of this failure is obtained by taking $e_1 = \mathbf{ncc}(\lambda k.k(k))$ and $e = \mathbf{set}(x, \mathbf{get}(x) + 1); y(\theta(x))$ where $\theta = \lambda x.\lambda d.\mathbf{get}(x)$. To see the problem, let

$$\begin{aligned} R_0 &= \mathbf{let } y = \bullet \mathbf{ in } \mathbf{set}(x, \mathbf{get}(x) + 1); y(\theta(x)) \\ e_l &= \mathbf{let } x = \mathbf{mk}(0) \mathbf{ in } R_0[\mathbf{ncc}(\lambda k.k(k))] \\ R_1 &= \mathbf{let } y = \bullet \mathbf{ in } \mathbf{let } x = \mathbf{mk}(0) \mathbf{ in } \mathbf{set}(x, \mathbf{get}(x) + 1); y(\theta(x)) \\ e_r &= R_1[\mathbf{ncc}(\lambda k.k(k))] \\ R &= \mathbf{if}(\mathbf{eq}(\bullet, 1), 1, \mathbf{Bot}(1)) \end{aligned}$$

then $(\diamond : R[e_r]) \downarrow$ but $\neg((\diamond : R[e_l]) \downarrow)$.

Semantically, an expression is control-free if in any computation context, evaluation of (any instance of) the expression uses no control rules. Syntactically, we can ensure this in Λ_s by requiring that \mathbf{ncc} does not appear in the expression (or any imported definitions), that \mathbf{get} does not appear (thus control effects can not be dynamically imported), and that all applications are of the form $\mathbf{app}(\lambda x.e, e')$ (so no functions with control effects can be imported from the environment). For example, $\mathbf{app}(\mathbf{get}(y), x)$ could import control

effects from the contents of y and $\mathbf{app}(w, x)$ could import control effects from the value of w , which is determined by the program context.

Lemma (delay.s): In Λ_s , if $x \notin \text{FV}(e_1)$, $y \notin \text{FV}(v)$, and e_1 is control free, then

$$(\mathbf{let} \ x = \mathbf{mk}(v) \ \mathbf{in} \ \mathbf{let} \ y = e_1 \ \mathbf{in} \ e) \cong (\mathbf{let} \ y = e_1 \ \mathbf{in} \ \mathbf{let} \ x = \mathbf{mk}(v) \ \mathbf{in} \ e)$$

Note that the argument used to establish **(delay)** in Λ_m now works, since **(mred)** holds for computations of allowed e_1 .

7 Uniform Computation

In this section we develop the machinery necessary to state precisely the **s-unif** property of uniform semantics. We first enrich λ -language syntax to include place-holders for expressions and reduction contexts. After stating the **s-unif** requirement, we show that Λ_s (and hence the contained languages) has uniform semantics by lifting the reduction rules for Λ_s to enriched states in a manner that meets the **s-unif** requirements.

To motivate our definition of enriched syntax, we first consider adding place-holders for closed expressions in the case where state contexts are trivial and states are simply expressions. The holes of traditional contexts (such as those we defined in §2) are a kind of place-holder for expressions. To keep the two uses separate we introduce \odot for place-holders. We add \odot to the clause defining expressions, just as we added \bullet to obtain standard contexts. Enriched versions of the remaining syntactic sorts are generated by replacing expressions by enriched expressions in the definitions. We signify entities of enriched syntax sorts by decorating metavariables with \star . Since \odot will only be replaced by closed expressions, we extend substitution to the enriched syntax, by defining $\odot^{\star\sigma} = \odot$. Since filling of place-holders by closed expression, e is the same as substitution of e for \odot , this simple enrichment is adequate for closed expressions. However, it doesn't provide an adequate notion of uniform reduction for arbitrary expressions. To see this consider $\star e = \mathbf{app}(\lambda y. \mathbf{app}(\lambda x. \lambda y. \odot, \lambda x. \odot), \lambda x. x)$. If we lift the reduction rules using the above definition of substitution, we have $\star e \longrightarrow \lambda y. \odot$ while $\star e[\odot \mapsto \mathbf{app}(x, y)] \longrightarrow \lambda y. \mathbf{app}(\lambda x. \mathbf{app}(x, \lambda x. x), y)$. Clearly we need to keep track of substitutions at occurrences of \odot . Thus we might try decorating \odot with a substitution to be carried out when the place-holder is filled and extend substitution to the enriched syntax by composing at place-holders. Starting with empty substitution decorating the occurrences of \odot in $\star e$ we have

$$\mathbf{app}(\lambda y. \mathbf{app}(\lambda x. \lambda y. \odot^\emptyset, \lambda x. \odot^\emptyset), \lambda x. x) \longrightarrow \mathbf{app}(\lambda x. \lambda y. \odot^\emptyset, \lambda x. \odot^{\{y \mapsto \lambda x. x\}})$$

and now we notice that to continue the computation, the decorating substitutions must be allowed to have values in the enriched syntax in their range.

Our solution to the problem of an appropriate notion of place-holder is based on idea of decorating holes with substitutions and it accomplishes several things. It provides a means of defining substitution on expressions enriched with place-holders in such a way that filling and substitution commute. This is the key to lifting the computation rules to enriched states. It also separates the mechanism for trapping free variables of the filling expression from the mechanism for binding of free variables in the filled expression. The former is the province of the decorating substitution, trapped variables are those in its domain. The latter is the province of λ as usual, and is propagated to the filling expression via binding of free variables in the range of the decorating substitution. A consequence of the separation is that alpha conversion is valid even for lambda variables with place-holders in their scope.

The presentation below is adapted from Agha, Mason, Smith, and Talcott (1997) where uniform computation methods are developed to establish equational laws for actor computations. This in turn was based on the theory of binding structures (Talcott 1991, 1993). We add E-holes (to be filled with expressions) to the summands of the defining equation for expressions, and R-holes (to be filled with reduction contexts) to the summands in the defining equation for reduction contexts. We also add R-holes with the redex hole filled to the summands of the defining equation for expressions. The specifications of the remaining syntactic classes are correspondingly modified to refer to the enriched syntax. We adopt the convention that an extended syntactic class is indicated by the mark $*$. Metavariables ranging over these classes are indicated by the same mark, and we prefix the names of these classes by ER-. Thus, we have ER-expressions where $*e$ ranges over $*\mathbf{E}$, ER-reduction contexts where $*R$ ranges over $*\mathbf{R}$, etc. Mostly we consider syntactic entities enriched with only on kind of place-holder. Thus one can read ER as E or R rather than E and R. We use the prefix E- and the mark \circ for E-hole enriched syntax and we use the prefix R- and the mark \diamond for R-hole enriched syntax. Thus we speak of E-expressions where $\circ e$ ranges over $\circ\mathbf{E}$ or R-expressions where $\diamond e$ ranges over $\diamond\mathbf{E}$.

For simplicity, we give the definitions for λ -languages in which the only additional values are pairs.

Definition ($*\mathbf{E}$, $*\mathbf{V}$, $*\mathbf{S}$, $*\mathbf{R}$, $*\mathbf{E}_{\text{rdx}}$):

$$*\mathbf{E} = \mathbf{A} \cup \mathbf{X} \cup \lambda\mathbf{X}.*\mathbf{E} \cup \mathbf{O}_n(*\mathbf{E}^n) \cup \circ^{*\mathbf{S}} \cup \diamond^{*\mathbf{S}}[*\mathbf{E}]$$

$$*\mathbf{V} = \mathbf{A} \cup \mathbf{X} \cup \lambda\mathbf{X}.*\mathbf{E} \cup \text{pr}(*\mathbf{V}, *\mathbf{V})$$

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

$$*\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{m+n+1}(*\mathbf{V}^m, *\mathbf{R}, *\mathbf{E}^n) \cup \diamond^{*\mathbf{S}}[*\mathbf{R}]$$

$$*\mathbf{E}_{\text{rdx}} = \mathbf{O}_n(*\mathbf{V}^n) - *\mathbf{V}$$

As before, λ is the only binding operator, and free variables of ER-expressions are defined as follows:

Definition (Free variables – $\text{FV}(*e), \text{FV}(*\sigma)$):

$$\begin{aligned} \text{FV}(\circ^{*\sigma}) &= \text{FV}(*\sigma) \\ \text{FV}(\diamond^{*\sigma}[*e]) &= \text{FV}(*\sigma) \cup \text{FV}(*e) \\ \text{FV}(x) &= \{x\} \\ \text{FV}(\lambda z.*e) &= \text{FV}(*e) - \{z\} \\ \text{FV}(\vartheta(*e_1, \dots, *e_n)) &= \text{FV}(*e_1) \cup \dots \cup \text{FV}(*e_n) \quad \vartheta \in \mathbf{O}_n \\ \text{FV}(*\sigma) &= \bigcup_{x \in \text{Dom}(*\sigma)} \text{FV}(*\sigma(x)) \end{aligned}$$

The variables in the domain of an occurrence of $*\sigma$ are neither free nor bound. Renaming of bound variables and substitution for free variables only act on the range of a substitution associated with a hole, not on its domain.

Definition (Substitution – $*e^{*\sigma}, *R^{*\sigma}, *\sigma \odot *\sigma'$): Substitution is extended to ER-expressions as follows:

$$\begin{aligned} (\circ^{*\sigma'})^{*\sigma} &= \circ^{*\sigma \odot *\sigma'} \\ (\diamond^{*\sigma'}[*e])^{*\sigma} &= \diamond^{*\sigma \odot *\sigma'}[*e^{*\sigma}] \\ x^{*\sigma} &= \begin{cases} x & \text{if } x \notin \text{Dom}(*\sigma) \\ *\sigma(x) & \text{if } *e \in \text{Dom}(*\sigma) \end{cases} \\ (\lambda z.*e)^{*\sigma} &= \lambda z. (*e^{*\sigma[(\text{Dom}(*\sigma) - \{z\})]}) \quad \text{if } z \notin \text{FV}(*\sigma[(\text{Dom}(*\sigma) - \{z\})]) \\ \vartheta(*e_1, \dots, *e_n)^{*\sigma} &= \vartheta(*e_1^{*\sigma}, \dots, *e_n^{*\sigma}) \\ \bullet^{*\sigma} &= \bullet \\ \diamond^{*\sigma'}[*R]^{*\sigma} &= \diamond^{*\sigma \odot *\sigma'}[*R^{*\sigma}] \\ *\sigma \odot *\sigma' &= \lambda x \in \text{Dom}(*\sigma'). *\sigma'(x)^{*\sigma} \end{aligned}$$

As defined here substitution is a partial operation. Using renaming substitutions (bijections on variables) we can rename bound variables in the usual way. We consider ER-expressions (and entities containing them) to be equal if they differ only by renaming of bound variables. Thus, for any substitution we can always choose a variant such that substitution is defined. Recall (§2) that such renaming is not possible in the case of traditional contexts.

The operations of filling E-holes in $*e$ with e , $*e[\circ \mapsto e]$, and filling R-holes in $*e$ with R , $*e[\diamond \mapsto R]$, are defined by induction on the structure of $*e$. As for substitution, we rename bound variables of $*e$ to avoid capture of free variables in e or R by lambda binding. When an expression is placed in an E-hole or a reduction context is placed in an R-hole the filled decorating substitution is applied to the filler. Similarly for filling of E- or R- holes in ER-reduction contexts.

Definition ($*e[\circ \mapsto e]$):

$$\circ^{*\sigma}[\circ \mapsto e] = e^{*\sigma[\circ \mapsto e]}$$

$$\begin{aligned}
\diamond^{*\sigma} [^*e][\circ \mapsto e] &= \diamond^{*\sigma[\circ \mapsto e]} [^*e[\circ \mapsto e]] \\
\vartheta(^*e_1, \dots, ^*e_n)[\circ \mapsto e] &= \vartheta(^*e_1[\circ \mapsto e], \dots, ^*e_n[\circ \mapsto e]) \\
^*v[\circ \mapsto e] &= ^*v \quad \text{if } ^*v \in \mathbf{A} \cup \mathbf{X} \\
(\lambda x. ^*e')[\circ \mapsto e] &= \lambda x. (^*e'[\circ \mapsto e]) \quad \text{if } x \notin \text{FV}(e) \\
^*\sigma[\circ \mapsto e] &= \lambda x \in \text{Dom}(^*\sigma). (^*\sigma(x)[\circ \mapsto e])
\end{aligned}$$

Consider the following simple example.

$$\begin{aligned}
(\lambda x. \circ^\emptyset)[\circ \mapsto x] &= (\lambda z. \circ^\emptyset)[\circ \mapsto x] = \lambda z. (\circ^\emptyset[\circ \mapsto x]) = \lambda z. x \\
(\lambda x. \circ^{x \mapsto x})[\circ \mapsto x] &= (\lambda z. \circ^{x \mapsto z})[\circ \mapsto x] = \lambda z. z = \lambda x. x
\end{aligned}$$

This example shows, among other things, that the precise domain of the substitution decorating a hole is important. This is in contrast to substitutions viewed as maps on expressions, where σ is the same map as $\sigma\{x \mapsto x\}$ for $x \notin \text{Dom}(\sigma)$.

Definition ($^*e[\diamond \mapsto R]$):

$$\begin{aligned}
\circ^{*\sigma}[\diamond \mapsto R] &= \circ^{*\sigma[\diamond \mapsto R]} \\
\diamond^{*\sigma} [^*e][\diamond \mapsto R] &= R^{*\sigma[\diamond \mapsto R]} [^*e[\diamond \mapsto R]] \\
\vartheta(^*e_1, \dots, ^*e_n)[\diamond \mapsto R] &= \vartheta(^*e_1[\diamond \mapsto R], \dots, ^*e_n[\diamond \mapsto R]) \\
^*v[\diamond \mapsto R] &= ^*v \quad \text{if } ^*v \in \mathbf{A} \cup \mathbf{X} \\
(\lambda x. ^*e')[\diamond \mapsto R] &= \lambda x. (^*e'[\diamond \mapsto R]) \quad \text{if } x \notin \text{FV}(R) \\
^*\sigma[\diamond \mapsto R] &= \lambda x \in \text{Dom}(^*\sigma). (^*\sigma(x)[\diamond \mapsto R])
\end{aligned}$$

As an example of R-hole filling we have

$$\diamond^{\{x \mapsto \lambda x. x\}} [2][\diamond \mapsto \mathbf{app}(x, \bullet)] = \mathbf{app}(\lambda x. x, 2)$$

It is easy (but tedious) to check that E-hole and R-hole filling commute.

Lemma (ER-hole filling):

$$^*e[\circ \mapsto e][\diamond \mapsto R] = ^*e[\diamond \mapsto R][\circ \mapsto e]$$

The following lemma is the key to developing a notion of uniform computation. A proof can be found in (Agha, Mason, Smith, and Talcott 1997).

Lemma (fil-subst): Hole filling and substitution commute.

$$\begin{aligned}
\circ^e \circ^\sigma[\circ \mapsto e'] &= \circ^e[\circ \mapsto e'] \circ^{\sigma[\circ \mapsto e']} \quad \text{if } \text{Dom}(\circ^\sigma) \cap \text{FV}(e') = \emptyset \\
\circ^e \circ^\sigma[\diamond \mapsto R] &= \circ^e[\diamond \mapsto R] \circ^{\sigma[\diamond \mapsto R]} \quad \text{if } \text{Dom}(\circ^\sigma) \cap \text{FV}(R) = \emptyset
\end{aligned}$$

Note that ER-reduction contexts possess two types of holes: decorated holes, and traditional, undecorated holes. The process of filling the redex hole, \bullet ,

with an E-expression, ${}^\circ e$, remains unchanged, and we continue to denote it by ${}^\circ R[{}^\circ e]$. Defining $\bullet[o \mapsto e] = \bullet[\diamond \mapsto R] = \bullet$ we see that filling of decorated and undecorated holes commutes.

Lemma (ER- C- hole fill):

$${}^*R[o \mapsto e_0][e] = {}^*R[e][o \mapsto e_0] \quad \text{and} \quad {}^*R[\diamond \mapsto e_0][e] = {}^*R[e][\diamond \mapsto e_0]$$

As an example of this commuting we have

$$\begin{aligned} & (\diamond^{\{x \mapsto \lambda x.x\}}[\bullet])[2][\diamond \mapsto \mathbf{app}(x, \bullet)] \\ &= (\diamond^{\{x \mapsto \lambda x.x\}}[\bullet])[\diamond \mapsto \mathbf{app}(x, \bullet)][2] = \mathbf{app}(\lambda x.x, 2) \end{aligned}$$

For each ordinary context C there is a corresponding E-expression, \widehat{C} such that

$$C[e] = \widehat{C}[o \mapsto e].$$

\widehat{C} is obtained by decorating each hole occurrence in C with a binding substitution $\{x_i \mapsto x_i \mid x_i \in X\}$ where X is the set of lambda variables having the hole occurrence in their scope. For example, the E-expression corresponding to $\lambda x.\bullet$ is $\lambda x.o^{x \mapsto x}$.

The decomposition lemma for ER-expressions has two new cases: when an E-hole or an R-hole appears in the redex position. As before, the proofs are an easy induction on the syntax structure.

Lemma (E-expression decomposition): For any ER-expression, *e , exactly one of the following holds:

- (0) ${}^*e \in {}^*\mathbf{V}$, or
- (1) $(\exists! {}^*R, {}^*r)({}^*e = {}^*R[{}^*r])$, or
- (2e) $(\exists! {}^*R, {}^*\sigma)({}^*e = {}^*R[o^{*\sigma}])$, or
- (2r) $(\exists! {}^*R, {}^*\sigma, {}^*v)({}^*e = {}^*R[\diamond^{*\sigma}[\bullet \mapsto {}^*v]])$

ER-states are composed of an ER-state context, and an ER-expression. ER-state contexts are formed just like state contexts replacing (value) expressions by ER- (value) expressions in the constructions. In particular, we continue to use ordinary holes and hole filling to represent state contexts and their conversion to expressions. Thus $\mathbf{s2e}(\zeta : {}^*e) = \zeta[{}^*e]$. An ER-state whose expression decomposes according to case (2e) or (2r) is said to *touch a hole*. E-hole filling of ER-states is defined by $(\zeta : {}^*e)[o \mapsto e] = \zeta[o \mapsto e] : ({}^*e[o \mapsto e])$. In the cases we have considered, E-hole filling of ER-state contexts is defined using the decomposition of states into a constant allocation part and an effects part. For example is the case of memory contexts we have

$$(\{z_i \mapsto \mathbf{mk}({}^\circ v_i) \mid 1 \leq i \leq n\})[o \mapsto e] = \{z_i \mapsto \mathbf{mk}({}^\circ v_i[o \mapsto e]) \mid 1 \leq i \leq n\}.$$

Similarly for R-hole filling.

The property required of ER-computation for uniform semantics can now be stated: an ER-state either reduces uniformly, hangs uniformly, or touches a hole.

Definition (Uniform reduction (s-unif)): A λ -language satisfies the **s-unif** property if the reduction rules can be extended to E-states and R-states such that:

1. If ${}^*\zeta : {}^*e \longrightarrow {}^*\zeta_1 : {}^*e_1$, then $({}^*\zeta : {}^*e)[\circ \mapsto e] \longrightarrow ({}^*\zeta_1 : {}^*e_1)[\circ \mapsto e]$ for any e and $({}^*\zeta : {}^*e)[\diamond \mapsto R] \longrightarrow ({}^*\zeta_1 : {}^*e_1)[\diamond \mapsto R]$ for any R .
2. If $({}^*\zeta : {}^*e)[\circ \mapsto e] \longrightarrow {}^*\zeta' : {}^*e'$ or $({}^*\zeta : {}^*e)[\diamond \mapsto R] \longrightarrow {}^*\zeta' : {}^*e'$, then either ${}^*\zeta : {}^*e$ touches a hole (*e has the form ${}^*R[P]$ where P is of the form $\circ^{*\sigma}$ or $\diamond^{*\sigma} [{}^*v]$) or there is some ${}^*\zeta_1, {}^*e_1$ such that ${}^*\zeta : {}^*e \longrightarrow {}^*\zeta_1 : {}^*e_1$.

We now show how to lift the reduction rules for the languages considered in the previous sections. Since place-holders can only occur in value expressions inside a lambda, their presence does not change the definition of the satisfaction relation for state contexts. Thus, the reduction rules for the various operations can be extended to the ER-redexes and ER-states simply by annotating metavariables with $*$'s. As examples, we give the local rules for **app**, **br**, and **eq**. the local lifting rule, and rules for **ncc** and **set**.

Definition (Reduction rules lifted):

$$\begin{aligned}
(\text{app}) \quad & \text{app}(\lambda x. {}^*e, {}^*v) \hookrightarrow_{*} {}^*e^{\{x \mapsto {}^*v\}} \\
(\text{br}) \quad & \text{br}({}^*v_0, {}^*v_1, {}^*v_2) \hookrightarrow_{*} \begin{cases} {}^*v_1 & \text{if } {}^*\zeta \models {}^*v \notin \{\text{nil}\} \\ {}^*v_2 & \text{if } {}^*\zeta \models {}^*v \in \{\text{nil}\} \end{cases} \\
(\text{eq}) \quad & \text{eq}({}^*v_0, {}^*v_1) \hookrightarrow_{*} \begin{cases} \mathbf{t} & \text{if } {}^*\zeta \models ({}^*v_0, {}^*v_1) \in \mathbf{AtEq} \\ \text{nil} & \text{if } {}^*\zeta \models ({}^*v_0, {}^*v_1) \notin \mathbf{AtEq} \end{cases} \\
(\text{rdx}) \quad & {}^*\zeta : {}^*R[{}^*r] \longrightarrow {}^*\zeta : {}^*R[{}^*e] \quad \text{if } {}^*r \hookrightarrow_{*} {}^*e \\
(\text{ncc}) \quad & {}^*\zeta : {}^*R[\text{ncc}({}^*v)] \longrightarrow {}^*\zeta : \text{app}({}^*v, \text{top} \circ {}^*R) \\
(\text{set}) \quad & {}^*M_T : {}^*R[\text{set}(z, {}^*v)] \longrightarrow {}^*M\{z \mapsto \text{mk}({}^*v)\}_T : {}^*R[\text{nil}] \\
& \text{if } z \in \text{Dom}({}^*M)
\end{aligned}$$

Theorem (Uniformity): The languages Λ_f , Λ_c , Λ_m , and Λ_s all have uniform semantics.

8 Proof of the **ciu** theorem and consequences

8.1 Proof of **ciu**

There are several proofs of the **ciu** theorem for Λ_m in the literature. The first proof (Mason and Talcott 1991a) uses the uniform computation technique,

but the details are not fully spelled out. In a more recent paper (Honsell, Mason, Smith, and Talcott 1995) a proof is given based on the observation that it suffices to show that \sqsubseteq^{ciu} is a congruence.

First we recall the statement of the theorem. The **ciu**-approximation relation is defined by

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

We want to show that for any λ -language with uniform semantics, $e_0 \sqsubseteq^{\text{ciu}} e_1$ iff $e_0 \sqsubseteq e_1$. That $e_0 \sqsubseteq e_1$ implies $e_0 \sqsubseteq^{\text{ciu}} e_1$ is (almost) direct from the definitions. It relies on the fact that

$$\zeta : R[e^\sigma] \uparrow \diamond : \zeta [R[e^\sigma]] \uparrow \diamond : \zeta [R[\text{let } \sigma \text{ in } e]]$$

where **let** σ **in** e expands the parallel substitution σ to a suitable sequence of **lets**, taking care that the sequentialization does not cause substitution into the range of σ . This fact follows from **g-unif**. Thus, we need only show that $e_0 \sqsubseteq e_1$ under the assumption that $e_0 \sqsubseteq^{\text{ciu}} e_1$. Using the representation of contexts as E-expressions it suffices to show that

$$(\zeta : \circ e)[\circ \mapsto e_0] \preceq (\zeta : \circ e)[\circ \mapsto e_1]$$

for any E-state $\zeta : \circ e$ such that both sides are closed. We use induction on the computation length. Assume $(\zeta : \circ e)[\circ \mapsto e_0] \downarrow$. First we consider the case in which e_0 is not a value expression. If $\circ e \in \circ \mathbf{V}$ then $(\zeta : \circ e)[\circ \mapsto e'] \downarrow$ for any e' since it is a value state. Otherwise, to show $(\zeta : \circ e)[\circ \mapsto e_1] \downarrow$, we need only find $\zeta' : \circ e'$ such that

1. $(\zeta : \circ e)[\circ \mapsto e_0] \longrightarrow (\zeta' : \circ e')[\circ \mapsto e_0]$ (in one or more steps)
2. $(\zeta' : \circ e')[\circ \mapsto e_1] \downarrow$ implies $(\zeta : \circ e)[\circ \mapsto e_1] \downarrow$.

since 1. together with the induction hypothesis imply $(\zeta' : \circ e')[\circ \mapsto e_1] \downarrow$.

If $\zeta : \circ e \longrightarrow \zeta' : \circ e'$ then we are done. Otherwise by **s-unif**, $\circ e$ has the form $\circ R[\circ^\sigma]$. Since e_0 is not a value, and holes in $\circ\sigma$ appear only inside lambdas it must be the case that e_0^σ has the form $\circ R_0[\circ r]$. Thus by **s-unif** there is some $\zeta' : \circ e'$ such that $\zeta : \circ R[e_0^\sigma] \longrightarrow \zeta' : \circ e'$. Clearly $\zeta' : \circ e'$ satisfies condition 1. To see that it satisfies condition 2. note that

$$(\zeta : \circ R[e_0^\sigma])[\circ \mapsto e_1] \longrightarrow (\zeta' : \circ e')[\circ \mapsto e_1]$$

and by the **ciu**-hypothesis

$$(\zeta : \circ R[e_0^\sigma])[\circ \mapsto e_1] \preceq (\zeta : \circ R[e_1^\sigma])[\circ \mapsto e_1].$$

For the case in which e_0 is a value expression, we prove a lemma (**uval**) that shows that by filling a finite number of holes in $\circ e$ with e_0 we obtain an

E-expression $\circ e_k$ such that either $\circ e_k$ is an E-value, or $\circ \zeta : \circ e_k$ reduces, thus giving $\circ \zeta' : \circ e'$ satisfying condition 1. Since the holes filled by e_0 appear in redex positions, we may apply the **ciu** assumption to show that 2. holds. We begin with the lemma.

Lemma (uval): If $(\circ \zeta : \circ e)[\circ \mapsto v] \downarrow$, then we can find $k \in \mathbf{N}$, $\circ r, \circ e_j, \circ R_j, \circ \sigma_j$ for $j \leq k$ such that

$$\begin{aligned} \circ e &= \circ e_0 \\ \circ e_j &= \circ R_j[\circ^{\circ \sigma_j}] \quad \text{for } j < k \\ \circ e_{j+1} &= \circ R_j[v^{\circ \sigma_j}] \end{aligned}$$

And either $\circ e_k$ is an E-value or there is some $\circ r$ such that $\circ e_k = \circ R_k[\circ r]$ and $\circ \zeta : \circ e_k$ reduces.

Proof (uval): Let $\circ e_0 = \circ e$ as required. By the decomposition lemma and **s-unif** either $k = 0$ or $\circ e$ has the form $\circ R_0[\circ^{\circ \sigma_0}]$. In the latter case, let $\circ e_1 = \circ R_0[v^{\circ \sigma_0}]$ and continue the process. Each step reduces the number of E-holes not in the scope of a lambda, and hence must terminate.

□_{uval}

Now let $k \in \mathbf{N}$, $\circ r, \circ e_j, \circ R_j, \circ \sigma_j$ for $j \leq k$ be as given by **(uval)** with $v = e_0$. Note that $\circ e_j[\circ \mapsto e_0] = \circ e[\circ \mapsto e_0]$ for $j \leq k$. Now we prove by induction on $k - j$ that $(\circ \zeta : \circ e_j)[\circ \mapsto e_1] \downarrow$ for $0 \leq j \leq k$. For $k = j$ we use the same argument as for the case in which e_0 is a non-value. Assume $(\circ \zeta : \circ e_{j+1})[\circ \mapsto e_1] \downarrow$ for some $j + 1 \leq k$. By construction

$$\circ e_{j+1}[\circ \mapsto e_1] = (\circ R_j[e_0^{\circ \sigma_j}])[\circ \mapsto e_1]$$

so by the **ciu** hypothesis

$$(\circ \zeta : \circ R_j[e_1^{\circ \sigma_j}])[\circ \mapsto e_1] \downarrow$$

which completes the e_0 is a value case.

□_{ciu}

We leave the proof of **(L-unif)** to the reader, but we note that ‘reduces assuming $p \in \mathbf{L}$ ’ can be reformulated by replacing free occurrences of p by $\lambda x. \circ^{\{x \mapsto x\}}$.

8.2 Proof of Equi-rcx

Now we use **ciu** combined with R-uniform computation to establish **(equi-rcx)**. For the readers convenience we first recall the statement of the theorem.

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

Remark In Λ_f or Λ_m establishing **(equi-rcx)** is a fairly straightforward application of **ciu** in combination with **(fred.isdef)** or **(mred.isdef)**. In Λ_c it is still fairly easy. However, to establish the result in a more general setting requires a bit more work. We use the R-hole uniformity of uniform semantics.

We prove a more general result.

Theorem (Equi-rcx!): For any λ -language with uniform semantics, to show $R_0[e] \cong R_1[e]$ for any e it suffices to show that for any $\zeta, \diamond R, \diamond \sigma, z$ fresh such that ζ closes $\diamond R[R_j[z]^{\diamond \sigma}]$ we can find an $\diamond e$ such that $\zeta : \diamond R[R_j[z]^{\diamond \sigma}] \longrightarrow \zeta : \diamond e$.

It is easy to see by uniformity properties that **(Equi-rcx!)** implies **(Equi-rcx)**. So we proceed with the proof of the more general result.

Proof (equi-rcx!): Assume the **(equi-rcx)** hypothesis. By **ciu**, we need only show that

$$\zeta : R[R_0[e]^\sigma] \downarrow \zeta : R[R_1[e]^\sigma]$$

for any closing ζ, R, σ . We claim that for any closing $\diamond \zeta, \diamond e$ if $(\diamond \zeta : \diamond e)[\diamond \mapsto R_0] \downarrow$, then $(\diamond \zeta : \diamond e)[\diamond \mapsto R_1] \downarrow$. Then taking $\zeta = \diamond \zeta$ and $\diamond e = R[\diamond^\sigma[e^\sigma]]$ we are done.

To prove the claim, pick some closing $\diamond \zeta, \diamond e$ such that $(\diamond \zeta : \diamond e)[\diamond \mapsto R_0] \downarrow$. If $\diamond e$ is an R-value or if $\diamond \zeta : \diamond e$ steps uniformly, then we are done. Otherwise, by the R-decomposition property $\diamond e$ has the form $\diamond R[\diamond^{\diamond \sigma}[\diamond v]]$. By assumption we can find $\diamond e'$ such that

$$\diamond \zeta : \diamond R[R_j^{\diamond \sigma}[\diamond v]] \longrightarrow \diamond \zeta : \diamond e' \quad \text{for } j < 2$$

and hence

$$(\diamond \zeta : \diamond e)[\diamond \mapsto R_j] \longrightarrow (\diamond \zeta : \diamond e')[\diamond \mapsto R_j] \quad \text{for } j < 2$$

If \longrightarrow is one or more steps for $j = 0$ or if $\diamond \zeta : \diamond e'$ steps uniformly, then

$$(\diamond \zeta : \diamond e')[\diamond \mapsto R_1] \downarrow \quad \text{by computation induction}$$

and we are done. Similarly, if $\diamond e' \in \diamond \mathbf{V}$ we are done. Thus by uniformity we may assume $\diamond e' = \diamond R[R_0^{\diamond \sigma}[\diamond v]]$ has the form $\diamond R'[\diamond^{\diamond \sigma'}[\diamond v']]$ and we apply the above argument to $\diamond \zeta : \diamond e'$. Since $\diamond e'$ has one fewer \diamond not in the scope of a lambda this process must terminate and, as in the proof of **ciu**, we are done.

□**Equi-rdx!**

9 Conclusion

In this paper we have unified our earlier work on semantics of imperative functional programs in a more abstract and general setting. We defined a notion of uniform semantics for λ -languages, and developed general principles for reasoning about program equivalence in any λ -languages with uniform

semantics. In particular we have shown that the **ciu** theorem holds in any λ -language with uniform semantics. For such languages we have the combined benefits of reduction calculi (modular axiomatization), and operational equivalence (more equations).

This is a small step towards a usable methodology for developing equational semantics of higher-order languages with effects, and there are a number of directions for future work. A useful refinement would be to identify a form of rules that guarantees uniform semantics, generalizing the ideas of Howe (1996) to functional languages with effects. Another extension would be to develop denotational tools in this setting generalizing Mason, Smith, and Talcott (1996).

Another direction is to treat a wider range of languages. Of particular interest is extending the uniform framework to incorporate actor and other concurrency primitives. As mentioned earlier, the notion of uniform computation was key to developing methods for reasoning about program equivalence in a λ -language with actor primitives. In the actor world termination is not an interesting property: it is the infinite (fair) computations that are of interest. Thus, rather than appealing to computation induction, we use uniform computation to transform computation paths, preserving fairness.

This more general setting should also facilitate extending the VTLoE programming logic (Honsell, Mason, Smith, and Talcott 1995) to a wider range of program primitives.

Acknowledgements

Most of the work that this paper builds on was done in collaboration with Ian Mason who contributed many important ideas and insights. The author would like to thank Ian Mason, Soeren Lassen, Scott Smith, José Meseguer, and Ian Stark for numerous helpful criticisms and pointing out errors in an earlier draft. She would also like to thank the editors of this volume, Andrew Gordon and Andrew Pitts for their encouragement and patience during the writing of the paper.

This research was partially supported by ARPA grant NAVY N00014-94-1-0775, ONR grant N00014-94-1-0857, NSF grant CCR-9302923, and NSF grant CCR-9312580.

References

- Abramsky, S. (1990). The lazy lambda calculus. In D. Turner (Ed.), *Research Topics in Functional Programming*. Addison-Wesley.
- Abramsky, S. (1991). Domain theory in logical form. *Annals of Pure and Applied Logic* 51(1), 1–77.

- Aczel, P. (1978, July). A generalized church-rosser theorem.
- Agha, G., I. A. Mason, S. F. Smith, and C. L. Talcott (1992, August). Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR '92)*, Volume 630 of *Lecture Notes in Computer Science*, pp. 565–579. Springer Verlag.
- Agha, G., I. A. Mason, S. F. Smith, and C. L. Talcott (1997). A foundation for actor computation. *Journal of Functional Programming*. to appear.
- Bloom, B. (1990). Can LCF be topped? *Information and Computation* 87, 264–301.
- Burge, W. H. (1975). Stream processing functions. *IBM Journal of Research and Development* 19, 12–25.
- Crank, E. and M. Felleisen (1991). Parameter-passing and the lambda-calculus. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 233–245.
- Egidi, L., F. Honsell, and S. Ronchi della Rocca (1992). Operational, denotational and logical descriptions: a case study. *Fundamenta Informaticae* 16(2), 149–170.
- Felleisen, M. (1987). *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph. D. thesis, Indiana University.
- Felleisen, M. (1988). λ -v-cs: An extended λ -calculus for Scheme. In *1988 ACM conference on Lisp and functional programming*, Volume 52, pp. 72–85.
- Felleisen, M. (1993). Personal communication.
- Felleisen, M. and D. Friedman (1986). Control operators, the SECD-machine, and the λ -calculus. In M. Wirsing (Ed.), *Formal Description of Programming Concepts III*, pp. 193–217. North-Holland.
- Felleisen, M. and R. Hieb (1989). The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University.
- Felleisen, M. and R. Hieb (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 235–271.
- Felleisen, M. and A. K. Wright (1991). A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University Computer Science Department. To appear, *Information and Computation*.
- Fields, J. and A. Sabry (1993). Reasoning about explicit and implicit representations of state. In *ACM Sigplan Workshop on State in Programming Languages*. YaleU/DCS/RR-968.
- Gordon, A. D. (1995, July). Bisimilarity as a theory of functional programming (mini-course). Technical Report NS-95-3, BRICS, Department of Computer Science, Aarhus University.
- Honsell, F., I. A. Mason, S. F. Smith, and C. L. Talcott (1995). A Variable Typed Logic of Effects. *Information and Computation* 119(1), 55–90.

- Howe, D. (1989). Equality in the lazy lambda calculus. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE.
- Howe, D. J. (1996). Proving congruence of bisimulation in functional programming languages. *Information and Computation* 124(2), 103–112.
- Jim, T. and A. Meyer (1991). Full abstraction and the context lemma. In *Theoretical Aspects of Computer Science*, Volume 526 of *Lecture Notes in Computer Science*, pp. 131–151. Springer-Verlag.
- Klop, J. (1980). *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam.
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal* 6, 308–320.
- Landin, P. J. (1966). The next 700 programming languages. *Comm. ACM* 9, 157–166.
- Lassen, S. B. (1995a). private communication, Oct 1995.
- Lassen, S. B. (1995b, December). Action semantics reasoning about functional programs. Technical Report NS-95-?, BRICS, Department of Computer Science, Aarhus University.
- Mason, I. A. (1986). *The Semantics of Destructive Lisp*. Ph. D. thesis, Stanford University. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.
- Mason, I. A. (1988). Verification of programs that destructively manipulate data. *Science of Computer Programming* 10, 177–210.
- Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From Operational Semantics to Domain Theory. *Information and Computation to appear*.
- Mason, I. A. and C. L. Talcott (1989). Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, Volume 372 of *Lecture Notes in Computer Science*, pp. 574–588. Springer-Verlag.
- Mason, I. A. and C. L. Talcott (1990). Reasoning about programs with effects. In *Programming Language Implementation and Logic Programming, PLILP'90*, Volume 456 of *Lecture Notes in Computer Science*, pp. 189–203. Springer-Verlag.
- Mason, I. A. and C. L. Talcott (1991a). Equivalence in functional languages with effects. *Journal of Functional Programming* 1, 287–327.
- Mason, I. A. and C. L. Talcott (1991b). Program transformation for configuring components. In *ACM/IFIP Symposium on Partial Evaluation and Semantics-based Program Manipulation*.
- Mason, I. A. and C. L. Talcott (1992). Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science* 105(2), 167–215.
- Mason, I. A. and C. L. Talcott (1994a). Program transformation via contextual assertions. In N. D. Jones, M. Hagiya, and M. Sato (Eds.), *Logic, Language,*

- and Computation: Festschrift in Honor of Satoru Takasu*, Number 792 in Lecture Notes in Computer Science, pp. 225–254. Springer-Verlag.
- Mason, I. A. and C. L. Talcott (1994b). Reasoning about object systems in vtloe. submitted to International Journal of Foundations of Computer Science.
- Milner, R. (1977). Fully abstract models of typed λ -calculi. *Theoretical Computer Science* 4, 1–22.
- Moggi, E. (1988). Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh.
- Moggi, E. (1989). Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE.
- Moggi, E. (1990). An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh.
- Morris, J. H. (1968). *Lambda calculus models of programming languages*. Ph. D. thesis, Massachusetts Institute of Technology.
- Mosses, P. D. (1992). *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
- Nipkow, T. (1991). Higher-order critical pairs. In *Sixth Annual Symposium on Logic in Computer Science*. IEEE.
- Ong, C.-H. (1988). *The Lazy Lambda Calculus: An investigation into the Foundations of Functional Programming*. Ph. D. thesis, Imperial College, University of London.
- Pitts, A. M. (1996). Reasoning about local variables with operationally-based logical relations. In *Eleventh Annual Symposium on Logic in Computer Science*. IEEE.
- Pitts, A. M. and I. Stark (1993). On the observable properties of higher order functions that dynamically create local names. In *ACM Sigplan Workshop on State in Programming Languages*. YaleU/DCS/RR-968.
- Pitts, A. M. and I. Stark (1996). Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*.
- Plotkin, G. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science* 1, 125–159.
- Rees, J. and W. e. Clinger (1986). The revised³ report on the algorithmic language scheme. *Sigplan Notices* 21(12), 37–79.
- Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Proceedings, ACM National Convention*, pp. 717–740.
- Ritter, E. and A. M. Pitts (1995). A fully abstract translation between a λ -calculus with reference types and standard ML. In *Proceedings TLCA'95, Edinburgh*.
- Sabry, A. and M. Felleisen (1993). Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* 6(3/4), 287–358.

- Smith, S. F. (1992). From operational to denotational semantics. In *MFPS 1991*, Volume 598 of *Lecture Notes in Computer Science*, pp. 54–76. Springer-Verlag.
- Steele, G. L. and G. J. Sussman (1975). Scheme, an interpreter for extended lambda calculus. Technical Report Technical Report 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory.
- Sullivan, G. T. (1996). *An Extensional MetaLanguage with I/O and a Dynamic Store*. Ph. D. thesis, Northeastern University. in preparation.
- Talcott, C. L. (1985). *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. Ph. D. thesis, Stanford University.
- Talcott, C. L. (1989). Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department.
- Talcott, C. L. (1991). Binding structures. In V. Lifschitz (Ed.), *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press.
- Talcott, C. L. (1992). A theory for program and data type specification. *Theoretical Computer Science* 104, 129–159.
- Talcott, C. L. (1993). A theory of binding structures and its applications to rewriting. *Theoretical Computer Science* 112, 99–143.
- Weeks, S. and M. Felleisen (1993). On the orthogonality of assignments and procedures in Algol. In *Proceedings 20th ACM Symposium on Principles of Programming Languages*, pp. 57–70.