The Coverage of Operational Semantics

Scott F. Smith
Department of Computer Science
The Johns Hopkins University
scott@cs.jhu.edu
http://www.cs.jhu.edu/scott/

Abstract

Techniques of operational semantics do not apply universally to all language varieties: techniques that work for simple functional languages may not apply to more realistic languages with features such as objects and memory effects. We focus mainly on the characterization of the so-called finite elements. The presence of finite elements in a semantics allows for an additional powerful induction mechanism. We show that in some languages a reasonable notion of finite element may be defined, but for other languages this is problematic, and we analyse the reasons for these difficulties.

We develop a formal theory of language embeddings and establish a number of properties of embeddings. More complex languages are given semantics by embedding them into simpler languages. Embeddings may be used to establish more general results and avoid reproving some results. It also gives us a formal metric to describe the gap between different languages.

Dimensions of the untyped programming language design space addressed here include functions, injections, pairs, objects, and memories.

1 Introduction

This paper is an exploration of a space of (untyped, deterministic) languages to determine what fundamental operational notions may be fruitfully defined.

A full and faithful notion of equivalence may be defined over an operational semantics via the Morris/Plotkin notion of operational/observational equivalence between program expressions, $\cong$. Operational equivalence has also been shown to yield good notions of equivalence for languages with memories (Mason and Talcott 1991), explicit control operators (Talcott 1989), types and objects (Gordon and Rees 1996), and distributed objects (Agha, Mason, Smith, and Talcott 1992).

It is important to characterize what "full and faithful" means: it is full in the sense that as many programs are equivalent as is possible, and it is faithful in the sense that we do not go overboard and equate programs that have differing behaviors. Operational equivalence $a \cong b$ is defined to precisely capture this notion: two program fragments are equivalent unless some use inside a larger program text can

distinguish the two. This program text can be viewed as a particular "test" of a. Letting C denote a program text and $C[a]$ the result of placing a in some "hole" in this text, we then may define $a \cong b$ as $C[a]$ and $C[b]$ having the same testing outcome. For many languages, termination suffices as the observation. Thus, to prove operational equivalence $a \cong b$, one supposes for arbitrary context C that $C[a]$ terminates and establishes that $C[b]$ also terminates, by induction on the size of $C[a]$'s computation (and, vice-versa).

It is a difficult question how operational equivalences $a \cong b$ may be established in practice (see (Talcott 1997) in this volume for more on this topic). Using computational induction as outlined above to directly establish operational equivalences is in fact very difficult. Even proving $1 + 1 \cong 2$ is difficult since a context could make many copies of $1 + 1$ (if for instance $1 + 1$ occurred in a function which was being passed to another function) before evaluating it. However, proofs are possible. A number of alternate characterizations of $\cong$ have been developed to make proofs of equivalence much simpler, including bisimulation (Abramsky 1990), applicative orderings (Bloom 1990), and **ciu** equivalence (Mason and Talcott 1991). These alternate characterizations may be shown to be the same as $\cong$ (a so-called *fully abstract* alternate equivalence). Bisimulation equivalences give rise to a coinduction principle which makes establishing equivalences easier; **ciu** equivalences still often must be established by computational induction, but the inductions are considerably simpler. Applicative orderings are very close in spirit to bisimulation orderings; in many settings the difference between the two could be called trivial.

The focus of this paper is the search for additional proof techniques for establishing fully abstract equivalences and other properties of programs. The main focus is the additional inductive structure of the finite or ω-algebraic elements in a domain. To review very briefly, the finite elements of a domain are the elements that are the lub of no infinite $\sqsubseteq$-directed set.

$$d \text{ is finite iff for all } \sqsubseteq\text{-directed sets } D \text{ with } d \sqsubseteq \bigsqcup D, d \sqsubseteq d_0 \text{ for some } d_0 \in D$$

Then, ω-algebraicity is the property that any domain element d can be decomposed into its finite elements:

$$\text{for all } d, d = \bigsqcup \{d_0 \mid d_0 \text{ is finite and } d_0 \sqsubseteq d\}$$

This equivalence allows a property of d to be proved by instead proving the property for each $d_0 \sqsubseteq d$ (assuming continuity also holds). If the finite elements also may be stratified into finite ranks $1, 2, \ldots, n, \ldots$, this gives rise to the possibility of proving properties of d by proving it for all $d_0 \sqsubseteq d$ by induction on the finite rank of d_0. This is very important because it gives a new, and often powerful, induction principle.

A closely related property is fixed point induction (Scott 1976). The greatest utility of the finite elements lies in the proof principle of rank induction, as just mentioned. For special cases of d it is possible to consider special forms of $\sqsubseteq$-ordered chain. One such example is fixed point induction, which is based on the

finite approximation set

$$\mathit{fix}(d) = \bigsqcup\{d(\bot), d(d(\bot)), \ldots, \overbrace{d(d(\ldots d(}^{n} \bot)\ldots), \ldots\}$$

for fixed point $\mathit{fix}(d)$. As in the case of finite elements, it is then possible to prove a property of the fixed point by proving a property of the finite approximants, by induction on n. So, even if ω-algebraicity cannot be established, there still may be particular finitary decompositions that lead to useful proof principles. Another example of the use of rank induction is found in the ideal model construction (MacQueen, Plotkin, and Sethi 1984), where it is used to give semantics to recursive types.

The obvious solution for defining finite elements is to work in a domain that models the language. The inductive structure is usually present in a domain directly by its manner of definition (Scott 1976). However, it is well-known that for many languages it is difficult to define a domain which has a notion of equivalence that is fully abstract. In fact, none of the languages studied in this paper have such a model extant in the literature. Our alternate approach here is to build finite elements out of the program syntax. This approach has proved successful for a particular simple functional language (Mason, Smith, and Talcott 1996). Syntactic projection functions $\pi^n(e)$ are defined within the programming language to project expression e to be a finite expression of level n. In this operational theory an additional powerful induction principle is thus obtained: induction on the rank of the finite elements.

Since memory-based languages often contain cyclical structures formed by memory self-references (for instance a function is in a cell and the function body contains a reference to the cell it is in), the idea of applying this technique to memory-based languages is particularly appealing. One problem in particular it could address is the question of a semantic definition of types in the presence of memory, a problem that is currently open.

So, the main goal of this paper is to address how the concept of finite elements generalizes to a broader class of languages. We will show that in some cases an effective finite element structure may be defined, and in other cases this cannot be established in a fully abstract manner (*i.e.*, finiteness may only be established with respect to an equivalence which is not operational equivalence $\cong$), showing the semantic tools which may be brought to bear on certain varieties of language are currently limited in this regard.

We want to consider a range of languages rather than addressing a single programming language. A series of (untyped) programming languages $\mathbb{L}$ is studied. $\mathbb{L}_{\mathrm{inj}}$ contains only the call-by-value λ-calculus and injections. $\mathbb{L}_{\mathrm{bpn}}$ adds booleans, numbers, and pairs. $\mathbb{L}_{\mathrm{obj}}$ has simple objects, and $\mathbb{L}_{\mathrm{m}}$ has a memory. We will not explicitly address control operators, typed languages, or concurrent or distributed computation. All languages we study follow the evaluation order most common in programming languages today, namely function application is call-by-value, pairing and injection are strict, and evaluation never takes place inside a λ.

Rather than presenting multiple semantic definitions, we give semantics to languages by defining language embeddings that map high-level languages down to low-level languages that lack many of the high-level features. In particular, all languages are mapped down to languages with injections only, $\mathbb{L}_{\text{inj}}$. This approach makes clear what the "difference" between languages is, and helps us focus on the particular difficulties that arise in some languages. A theory of embeddings is developed and a number of theorems proved to better characterize what can be embedded in what and how well. The mappings also allow us to define some hybrid forms of language quite easily, and in certain cases allows the "lifting" of theorems from low-level to high-level languages. The approach of defining language features by embeddings has a long history going back to Strachey (Milne and Strachey 1976). We obtain some interesting results about these mappings, and the mappings themselves are an additional topic of the paper.

An outline of the paper is as follows. A language-independent framework for operational semantics and language embeddings is defined in Section 2. Next, in Section 3 the injection language $\mathbb{L}_{\text{inj}}$ is studied in detail and its finite element theory developed. Then, in Sections 4 and 5 a wider space of languages is explored via embeddings into $\mathbb{L}_{\text{inj}}$. Section 5 presents a memory language $\mathbb{L}_{\text{m}}$ in full detail. Conclusions regarding what succeeds and what fails are found in Section 6.

2 A Framework for Operational Semantics

Before studying particular languages we define a simple semantic framework for languages with an operational evaluation relation, and for embeddings between languages. It is general enough to encompass all languages studied herein, but is not intended as a general framework along the lines of (Mosses 1992). In particular it will not fully capture nondeterministic or concurrent languages. It will allow for a general notion of language embedding to be defined. We will give definitions and properties that hold over an arbitrary operational structure; these definitions will then not have to be repeated for each language studied.

2.1 Operational Structures

We begin with an official definition of a language structure, called an *operational structure*. Languages are taken to come with an operational evaluation relation. Some of the language mappings need to be based on the grammatical structure of the language, so a general notation for operators $\mathtt{op}$ is also defined in the tradition of a theory of arities (Harper, Honsell, and Plotkin 1993). All languages are defined with respect to a single shared set of program variables $\mathbb{X}$ for simplicity.

DEFINITION 2.1 A Language $\mathbb{L}$ has structure $\langle \mathbb{E}, \mathbb{V}, \mathbb{O}, \mapsto \rangle$ where

$\mathbb{E}$ is the set of expressions of the language

$\mathbb{V} \subseteq \mathbb{E}$ are the value expressions

$\mathbb{O}$ are the operators. Each operator $\mathtt{op} \in \mathbb{O}$ can be viewed as a map from $(\mathbb{X}^{m_1} \times \mathbb{E}) \times \ldots \times (\mathbb{X}^{m_n} \times \mathbb{E})$ to $\mathbb{E}$ for some values of n and m_i, $1 \leq i \leq n$ associated with $\mathtt{op}$.

$\mapsto \in \mathbb{E} \times \mathbb{E}$ is the evaluation relation, mapping expressions to final computation results. It is reflexive on values.

Our notion of operational evaluation relation maps expressions to expressions; although this may seem restrictive, it is possible to define evaluation relations for languages with control primitives and effects in a purely syntactic fashion (Mason and Talcott 1991; Felleisen and Hieb 1992). We impose some informal regularity on the notation used for languages. $\mathbb{L}$ is implicitly $\langle \mathbb{E}, \mathbb{V}, \mathbb{O}, \mapsto \rangle$, $\mathbb{L}_{\mathrm{inj}(i)}$ taken to be $\langle \mathbb{E}_{\mathrm{inj}(i)}, \mathbb{V}_{\mathrm{inj}(i)}, \mathbb{O}_{\mathrm{inj}(i)}, \mapsto_{\mathrm{inj}(i)} \rangle$, $\mathbb{L}'$ taken to be $\langle \mathbb{E}', \mathbb{V}', \mathbb{O}', \mapsto' \rangle$, etc. We let x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, a, b, c, d, e range over $\mathbb{E}$, and $\mathtt{op}$ range over $\mathbb{O}$. For the remainder of this section we develop results for an arbitrary fixed language $\mathbb{L}$. Each definition of this section, *e.g.* $\cong$, is applied to a particular language by subscripting: $\cong_{\mathrm{m}}$ indicates the $\cong$ relation for language $\mathbb{L}_{\mathrm{m}}$.

The operators are a general notation which allows each field to bind some number of variables. In each product $(\mathbb{X}^{m_i} \times \mathbb{E})$, the variables bind free occurrences in the expression. An example operator is

$$\mathtt{sample-op}(x.y.e_1, e_2, z.e_3)$$

—free x and y in e_1 and free z in e_3 are bound, and no other free variables in e_1, e_2, e_3 are bound. We will use this informal notation of writing out an example to define the arity of an operator.

The set of expressions $\mathbb{E}$ is constrained to be the least superset of $\mathbb{X}$ closed under the operators in $\mathbb{O}$. We will implicitly coerce between operators of one language and operators of another language, provided the arities of the two operators are the same. Furthermore, if an expression $e \in \mathbb{L}_1$ is constructed with operators which are all in $\mathbb{O}_2$ for some $\mathbb{L}_2$, then e may be implicitly coerced to be in $\mathbb{E}_2$ by the obvious pointwise operator mapping.

A *closed expression* is an expression with no free variables; $\mathbb{E}^{\emptyset}$ is the set of all closed expressions. $a\,[b/x]$ is the result of substituting b for the free occurrences of x in a taking care not to trap free variables of b. *Contexts* $C \in \mathbb{C}$ are expressions with holes "$\bullet$" punched in them, and $C[e]$ denotes placing e in the hole(s) in C, possibly incurring the capture of some free variables in e.

A *value substitution* is a finite map from variables to values. We let σ range over value substitutions. $a\,[\sigma]$ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in a by $\sigma(x)$, again taking care not to trap variables.

2.2 Operational Ordering and Equivalence

In this section we give basic definitions of orderings and equivalence that are uniform with respect to the language studied.

DEFINITION 2.2 ($\lesssim$, $\cong$)

$a \lesssim b$ iff for all $C \in \mathbb{C}$ such that $C[a], C[b] \in \mathbb{E}^{\emptyset}, C[a]\downarrow$ implies $C[b]\downarrow$
$a \cong b$ iff $a \lesssim b$ and $b \lesssim a$

Note, a is *defined* (written $a\downarrow$) if it evaluates to a result: $a \mapsto b$ for some b. And, $a\uparrow$ if $a\downarrow$ fails to hold. In this definition of observational equivalence we are implicitly taking termination as the single observable property because it is a proper notion of observation for the languages studied herein. A more complete treatment would allow for a more general notion of observation than just termination.

LEMMA 2.3 (ELEMENTARY $\lesssim$ / $\cong$ PROPERTIES) (i) $\lesssim$ is transitive and reflexive (a pre-order).

(ii) $\cong$ is an equivalence relation.

(iii) $\lesssim$ is a pre-congruence, i.e. $a \lesssim b$ implies $C[a] \lesssim C[b]$.

(iv) $\cong$ is a congruence, i.e. $a \cong b$ implies $C[a] \cong C[b]$.

Over any operational structure, it is also possible to define an operational notion of directed set. This is simply a $\lesssim$-directed set of expressions in place of a $\sqsubseteq$-directed set of domain elements.

DEFINITION 2.4 ($\lesssim$-DIRECTED SETS) A set A is *directed* iff for all a, b, if $a, b \in A$, then there is some $c \in A$ where $a \lesssim c$ and $b \lesssim c$.

We let A, B range over directed sets with finitely many free variables[1], and V range over directed sets with $V \subseteq \mathbb{V}$. We allow directed sets of expressions to be used as subexpressions with the convention $C[A] = \{C[a] \mid a \in A\}$. Value substitutions σ extend pointwise to sets of expressions: $A\,[\sigma] = \{a\,[\sigma] \mid a \in A\}$.

The first hurdle encountered is the lack of an operational analogue of a lub operator $\sqcup$. Some chains may not even have an upper bound because each element of the chain could be a function with a finite domain, but the lub could have an infinite domain and be uncomputable. Thus, $\lesssim$ is not complete.

A number of solutions to this problem are possible. For one, we could restrict the directed sets A to be recursively enumerable (r.e.) sets. The lub will also be r.e. since A is r.e. This approach may be effectively applied to the simply typed λ-calculus (Freyd, Mulry, Rosolini, and Scott 1990).

[1] For technical reasons, we only allow directed sets with finitely many free variables, otherwise a directed set may contain all the variables $\mathbb{X}$ free and problems may arise in obtaining fresh variables.

A variation would be to further require that the directed set A is *internally* represented by a function f in $\mathbb{L}$ such that $f(n)$ for natural number n produces the n-th element of the directed set, and to have an expression $\texttt{lub}(f)$ in the language which internally computes the lub of f (this argument assumes the language $\mathbb{L}$ has functions and numbers). Unfortunately, `lub` must then have the ability to dovetail computations in the directed set, and this will require new syntax in the language and change the underlying equivalence. So, this approach may be of limited value.

It thus does not appear to be feasible to construct structures isomorphic to domains directly on expressions. It is however possible to make progress by using $\underset{\sim}{\sqsubseteq}$-directed sets of expressions as a space over which an ordering is defined (Smith 1992). A simple pre-ordering on directed sets of expressions, $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$, is defined for this purpose. This pre-ordering has the property that $a \underset{\sim}{\sqsubseteq} b$ iff $\{a\} \underset{\sim}{\sqsubseteq} \{b\}$, meaning that it fully and faithfully generalizes $\underset{\sim}{\sqsubseteq}$.

DEFINITION 2.5 (SET RELATIONS $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$, $\{\cdot\} \cong \{\cdot\}$) For A, B directed, define

$$\begin{array}{lll} A \underset{\sim}{\sqsubseteq} B & \text{iff} & \text{for all } a \in A \text{ and for all } C \in \mathbb{C} \text{ such that } C[A], C[B] \subseteq \mathbb{E}^{\emptyset}, \\ & & \text{if } C[a]\downarrow \text{ then there exists a } b \in B \text{ such that } C[b]\downarrow. \\ A \cong B & \text{iff} & A \underset{\sim}{\sqsubseteq} B \text{ and } B \underset{\sim}{\sqsubseteq} A \end{array}$$

This ordering is not quite what one might initially expect, as the $b \in B$ may be chosen depending on the particular testing context C which exercises $a \in A$. Therein lies the power of the ordering. From the context of use it will be possible to disambiguate between $\underset{\sim}{\sqsubseteq}$ and $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$. Some elementary properties of $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$ include the following.

LEMMA 2.6 (ELEMENTARY $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}/\{\cdot\} \cong \{\cdot\}$ PROPERTIES) (i) $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$ is a pre-congruence: $A \underset{\sim}{\sqsubseteq} B$ implies $C[A] \underset{\sim}{\sqsubseteq} C[B]$.

(ii) $\{\cdot\} \cong \{\cdot\}$ is a congruence: $A \cong B$ implies $C[A] \cong C[B]$.

(iii) $\{a\} \underset{\sim}{\sqsubseteq} \{b\}$ iff $a \underset{\sim}{\sqsubseteq} b$.

(iv) $A \underset{\sim}{\sqsubseteq} \{b\}$ iff for all $a \in A, a \underset{\sim}{\sqsubseteq} b$.

(v) $a \in A$ implies $\{a\} \underset{\sim}{\sqsubseteq} A$.

This ordering will be used to characterize the finite expressions.

DEFINITION 2.7 (FINITE EXPRESSIONS $\mathbb{E}^{\omega}$) The set of finite expressions $\mathbb{E}^{\omega}$ is defined by

$$\mathbb{E}^{\omega} = \{b \in \mathbb{E}^{\emptyset} \mid \text{for all closed } A, \text{ if } \{b\} \underset{\sim}{\sqsubseteq} A \text{ then } b \underset{\sim}{\sqsubseteq} a \text{ for some } a \in A\}$$

It is worth emphasizing that the previous development was language-independent: the ordering $\{\cdot\} \underset{\sim}{\sqsubseteq} \{\cdot\}$ was successfully defined over any language $\mathbb{L}$. Much of the power of this relation is derived from its generality. This completes the brief general theory of languages. We next define a theory of embeddings between languages.

2.3 Language Embeddings

For the purpose of embedding one programming language in another, a general theory of language embeddings is now defined. Other closely related notions of language embedding have been previously defined (Felleisen 1991; Mitchell 1993; Riecke 1993). Our definitions combine ideas from these approaches. The embeddings will be used to give semantics to a number of languages in Section 4, and will enable a number of properties to be proved concerning the relation between equivalences in one language and equivalences in a similar language.

DEFINITION 2.8 (EMBEDDING) Given high- and low-level languages $\mathbb{L}_h$ and $\mathbb{L}_l$, a sound embedding $[\![\cdot]\!] \in \mathbb{L}_h \to \mathbb{L}_l$ is defined as maps $[\![\cdot]\!] \in \mathbb{E}_h \to \mathbb{E}_l$ $[\![\cdot]\!] \in \mathbb{C}_h \to \mathbb{C}_l$ and an initial context $C_{\text{init}} \in \mathbb{C}_l$ such that

(i) For $C \in \mathbb{C}_h, e \in \mathbb{E}_h$, $[\![C[e]]\!] = [\![C]\!][[\![e]\!]]$

(ii) For closed $e \in \mathbb{E}_h$, $e \downarrow_h$ iff $C_{\text{init}}[[\![e]\!]] \downarrow_l$

C_{init} is not $\bullet$ for embeddings that require a special initial context; for most embeddings, C_{init} will just be $\bullet$. Our (i) is Mitchell's (R1) condition, and (ii) is similar to Felleisen's condition 2 of his Eliminability notion (Felleisen 1991).

For brevity in this presentation, we will generally not define $\downarrow_h$ and $\mapsto_h$ for the high-level languages. We will often leave $\mapsto_h$ undefined and take (ii) above as a *definition* of $\downarrow_h$. Since operational equivalence needs only to have termination defined, termination alone is a sufficient operational characterization for our purposes. Sound embeddings will thus only need to satisfy (i). Value set definitions $\mathbb{V}_h$ will similarly not be needed. We take $[\![A]\!]$ to abbreviate $\{[\![a]\!] \mid a \in A\}$.

It is useful to consider the induced ordering $[\![a]\!] \sqsubseteq_l [\![b]\!]$. Since programs in $\mathbb{L}_h$ will be able to be tested by contexts that are not in the codomain of the translation, this equivalence may be more fine-grained than $\sqsubseteq_h$.

LEMMA 2.9 If $[\![a]\!] \sqsubseteq_l [\![b]\!]$ then $a \sqsubseteq_h b$; and, if $[\![A]\!] \sqsubseteq_l [\![B]\!]$ then $A \sqsubseteq_h B$.

PROOF: For arbitrary C, suppose $C[a] \downarrow_h$, show $C[b] \downarrow_h$, assuming $[\![a]\!] \sqsubseteq_l [\![b]\!]$. Since the embedding is sound, $[\![C[a]]\!] = [\![C]\!][[\![a]\!]]$ and $C_{\text{init}}[[\![C]\!][[\![a]\!]]] \downarrow_l$ and thus by assumption $C_{\text{init}}[[\![C]\!][[\![b]\!]]] \downarrow_l$, allowing us to conclude $C[b] \downarrow_h$. The proof for the set-based ordering is similar, noting that if $[\![A]\!]$ is directed, A also is. □

An even more desirable property of embeddings is *full abstraction*, Mitchell's (R2). This is the case when the $\mathbb{L}_l$ contexts cannot expose any more structure than the $\mathbb{L}_h$ contexts already had exposed.

DEFINITION 2.10 An embedding $[\![\cdot]\!] \in \mathbb{L}_h \to \mathbb{L}_l$ is *fully abstract* if $a \sqsubseteq_h b$ iff $[\![a]\!] \sqsubseteq_l [\![b]\!]$. It is *set fully abstract* if $A \sqsubseteq_h B$ iff $[\![A]\!] \sqsubseteq_l [\![B]\!]$.

Set full abstraction trivially implies full abstraction by the definitions, but not the converse. (It is an open question whether the converse holds).

Full abstraction imposes a strong global structure on the embedding, one that many embeddings will fail to satisfy. It is also useful to consider imposing local structure on the embedding. The first additional constraint we impose is the notion of a *parametric* embedding.

DEFINITION 2.11 (PARAMETRIC EMBEDDING) Given a sound embedding $[\![\cdot]\!] \in \mathbb{L}_h \to \mathbb{L}_l$, it is *parametric* if for each $\mathtt{op} \in \mathbb{O}_h$, an expression $e_{\mathtt{op}}$ with $\mathrm{FreeVars}(e_{\mathtt{op}}) = \{y_1, \ldots, y_n\}$ exists such that

$$[\![\mathtt{op}(\overline{x_1}.e_1, \ldots, \overline{x_n}.e_n)]\!] = e_{\mathtt{op}}\,\{[\![e_1]\!]/y_1,\ldots,[\![e_n]\!]/y_n\}$$

where $a\,\{b/x\}$ indicates a substitution of all free occurrences of variable x by b, *allowing* the free variables of b to be captured in a. Furthermore, there is a $\mathbb{L}_l$ context C_{var} such that

$$[\![x]\!] = C_{\text{var}}[x]$$

C_{var} interprets variables; in more complex embeddings such as state and control, variables cannot just map to themselves. The reader may now want to look ahead to Definitions 4.3, 4.18, and 5.3, which contain examples of parametric embeddings. Parametric embeddings are closely related to Felleisen's condition 3, so-called "Macro Eliminability" (Felleisen 1991).

One other important property of an embedding is the syntax of some high-level operators may be preserved by the embedding.

DEFINITION 2.12 (HOMOMORPHIC EMBEDDING) Given a parametric embedding $[\![\cdot]\!] \in \mathbb{L}_h \to \mathbb{L}_l$, this embedding is a *homomorphic embedding* with respect to operators $\mathbb{O}$ if

(i) $C_{\text{init}} = \bullet$

(ii) $\mathbb{O} \subseteq \mathbb{O}_h \cap \mathbb{O}_l$, and the arity of operators in $\mathbb{O}$ is the same in $\mathbb{O}_l$ and $\mathbb{O}_h$

(iii) for each $\mathtt{op} \in \mathbb{O}$, the embedding is homomorphic:

$$[\![\mathtt{op}(\overline{x_1}.e_1, \ldots, \overline{x_m}.e_n)]\!] = \mathtt{op}(\overline{x_1}.[\![e_1]\!], \ldots, \overline{x_m}.[\![e_n]\!])$$

(iv) $C_{\text{var}} = \bullet$

In terms of equivalence, homomorphic embeddings preserve equations that only use operators in $\mathbb{O}$. This allows equations in the low-level language to be lifted to the higher language, avoiding the need to re-prove them.

LEMMA 2.13 (LIFTING) If $a \precsim_l b$ and a and b only use operators in $\mathbb{O}$, and $[\![\cdot]\!] \in \mathbb{L}_h \to \mathbb{L}_l$ is a homomorphic embedding with respect to $\mathbb{O}$, then $a \precsim_h b$. Similarly, $A \precsim_l B$ implies $A \precsim_h B$ when A and B only use operators in $\mathbb{O}$.

PROOF: $[\![a]\!] = a$ and $[\![b]\!] = b$ in this case; the result then follows from the fact that all testing contexts $C \in \mathbb{C}_h$ can be mapped to $[\![C]\!] \in \mathbb{C}_l$. □

The main advantage of homomorphic embeddings is the abilty to use the Lifting Lemma. Our notion of homomorphic embedding is closely related to Felleisen's eliminability definition, condition 1 (Felleisen 1991). Definitions 4.3 and 4.18 below are examples of homomorphic embeddings.

Embeddings compose in the obvious fashion.

LEMMA 2.14 Define $[\![\cdot]\!] \in \mathbb{L}_1 \to \mathbb{L}_3$ as the composition of $[\![\cdot]\!]_1 \in \mathbb{L}_1 \to \mathbb{L}_2$ and $[\![\cdot]\!]_2 \in \mathbb{L}_2 \to \mathbb{L}_3$ formed by defining $[\![\cdot]\!] \in \mathbb{E}_1 \to \mathbb{E}_3$ as $[\![\cdot]\!]_2 \circ [\![\cdot]\!]_1$.

(i) $[\![\cdot]\!] \in \mathbb{L}_1 \to \mathbb{L}_3$ is a sound embedding.

(ii) If $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$ are both parametric (resp., homomorphic with respect to $\mathbb{O}_1$ and $\mathbb{O}_2$) embeddings, then $[\![\cdot]\!]$ is also a parametric (resp., homomorphic with respect to $\mathbb{O}_1 \cap \mathbb{O}_2$) embedding.

3 The Injection Languages $\mathbb{L}_{\mathrm{inj}(k)}$

The development up to this point has been independent of a particular language. We now study a particular family of languages in-depth, the family of injection languages $\mathbb{L}_{\mathrm{inj}(1)}, \mathbb{L}_{\mathrm{inj}(2)}, \ldots$. Following this section, we define a series of languages via embeddings into $\mathbb{L}_{\mathrm{inj}(k)}$. Pure λ-expressions plus injections provide just enough syntax to allow a wide range of programming languages to be faithfully embedded. The injections $\mathtt{inj}_i(a)$ serve to wrap high-level expressions a to distinguish different sorts; Lemma 4.5 below shows why the pure λ-calculus will not suffice as an embedding target language. The injection languages consist of the untyped call-by-value λ-calculus augmented with a finite number of injections. The language $\mathbb{L}_{\mathrm{inj}(k)}$ has injections $\mathtt{inj}_1(a), \ldots, \mathtt{inj}_k(a)$.

DEFINITION 3.1 ($\mathbb{L}_{\mathrm{inj}(k)}$) For $k \in \mathbb{N}$, define

$$\mathbb{L}_{\mathrm{inj}(k)} = \langle \mathbb{E}_{\mathrm{inj}(k)}, \mathbb{V}_{\mathrm{inj}(k)}, \mathbb{O}_{\mathrm{inj}(k)}, \mapsto_{\mathrm{inj}(k)} \rangle$$

as follows.

$$\mathbb{O}_{\mathrm{inj}(k)} = \{\mathtt{app}(a,b), \mathtt{lambda}(x.a)\} \cup \bigcup_{1 \le i \le k} \{\mathtt{inj}_i(a), \mathtt{out}_i(a), \mathtt{is}_i(a)\}$$

$$\mathbb{V}_{\mathrm{inj}(k)} = \mathbb{X} \cup \{\mathtt{lambda}(x.a) \mid a \in \mathbb{E}_{\mathrm{inj}(k)}\} \cup \bigcup_{1 \le i \le k} \{\mathtt{inj}_i(v) \mid v \in \mathbb{V}_{\mathrm{inj}(k)}\}$$

$\mapsto_{\mathrm{inj}(k)}$ is defined below.

For the remainder of this section, the k in $\mathbb{L}_{\mathrm{inj}(k)}$ is taken to be some arbitrary fixed value and we write $\mathbb{L}_{\mathrm{inj}}$, $\mathbb{E}_{\mathrm{inj}}$, etc. Several syntactic abbreviations will be made

to aid in readability of programs through the use of more conventional notation. These include

$$\begin{array}{rcl}
\lambda x.a & = & \texttt{lambda}(x.a) \\
a\texttt{(}b\texttt{)} & = & \texttt{app}(a,b) \\
\texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b & = & (\lambda x.b)\texttt{(}a\texttt{)} \\
\texttt{bot} & = & (\lambda x.x\texttt{(}x\texttt{)})\texttt{(}\lambda x.x\texttt{(}x\texttt{)}\texttt{)} \\
\texttt{etrue} & = & \lambda x.\lambda y.x\texttt{(}x\texttt{)} \\
\texttt{efalse} & = & \lambda x.\lambda y.y\texttt{(}y\texttt{)} \\
\texttt{eif}(a,b,c) & = & a\texttt{(}\lambda x.b\texttt{)(}\lambda x.c\texttt{)} \text{ for fresh } x \\
\texttt{fix} & = & \lambda y.(\lambda x.\lambda z.y\texttt{(}x\texttt{(}x\texttt{)}\texttt{)(}z\texttt{)})\texttt{(}\lambda x.\lambda z.y\texttt{(}x\texttt{(}x\texttt{)}\texttt{)(}z\texttt{)}\texttt{)} \\
a \circ b & = & \lambda x.a\texttt{(}b\texttt{(}x\texttt{)}\texttt{)}
\end{array}$$

Booleans and conditional are encoded via the standard method. The "e" prefixing $\texttt{etrue}, \texttt{efalse}, \texttt{eif}$ indicates that these are encoded notions of these constructs and not primitives. Observe typewriter font parentheses $a\texttt{(}b\texttt{)}$ abbreviate function application. `fix` is a call-by-value version of the standard fixed-point combinator for functionals.

3.1 Operational Semantics

The operational semantics of expressions is given by a single-step evaluation relation $\mapsto_{\text{inj}}$, using the convenient notion of a reduction context (a.k.a. evaluation context) from (Felleisen, Friedman, and Kohlbecker 1987). The *redices* are expressions of the form $v\texttt{(}a\texttt{)}$, $\texttt{is}_i(v)$, or $\texttt{out}_i(v)$. Redices are either immediately available for execution, $\texttt{out}_i(\texttt{inj}_i(a))$, or are *stuck*, $\texttt{out}_i(\lambda x.x)$. In this presentation stuck computations are treated as divergent for simplicity. Reduction contexts $\mathbb{R}_{\text{inj}}$ determine the subexpression that is to be reduced next.

DEFINITION 3.2 (REDUCTION CONTEXTS $\mathbb{R}_{\text{inj}}$) The set of reduction contexts, $R \in \mathbb{R}_{\text{inj}}$, is the least subset of $\mathbb{C}_{\text{inj}}$ that includes

$$\bullet, R\texttt{(}e\texttt{)}, v\texttt{(}R\texttt{)}, \texttt{inj}_i(R), \texttt{out}_i(R), \texttt{is}_i(R)$$

for all $e \in \mathbb{E}_{\text{inj}}$, $v \in \mathbb{V}_{\text{inj}}$, $R \in \mathbb{R}_{\text{inj}}$, and $1 \leq i \leq k$.

In an expression $R[a]$, R denotes the continuation for the computation a. Reduction contexts are used in evaluation as follows. In order to perform one step of computation of some non-value expression a, it is uniquely decomposed into $a = R[b]$ for some R and redex b by the following Lemma. Uniqueness of decomposition implies evaluation is deterministic.

LEMMA 3.3 (DECOMPOSITION) Either $a \in \mathbb{V}_{\text{inj}}$ or a can be written uniquely as $R[b]$ where b is a redex.

DEFINITION 3.4 (EVALUATION $\mapsto_{\text{inj}}$) The evaluation relation $\mapsto_{\text{inj}}$ for $\mathbb{L}_{\text{inj}(k)}$ is the transitive, reflexive closure of the single-step evaluation relation $\mapsto^1_{\text{inj}}$, which is generated by the following clauses:

$$\begin{array}{rrcl}
\text{(beta)} & R[(\lambda x.a)\,(v)] & \mapsto^1_{\text{inj}} & R[a\,[v/x]] \\
\text{(out)} & R[\mathtt{out}_i(\mathtt{inj}_i(v))] & \mapsto^1_{\text{inj}} & R[v] \\
\text{(is-t)} & R[\mathtt{is}_i(\mathtt{inj}_i(v))] & \mapsto^1_{\text{inj}} & R[\mathtt{etrue}] \\
\text{(is-f)} & R[\mathtt{is}_i(\mathtt{inj}_j(v))] & \mapsto^1_{\text{inj}} & R[\mathtt{efalse}] \text{ where } i \neq j \\
\text{(is-lam)} & R[\mathtt{is}_i(\lambda x.a)] & \mapsto^1_{\text{inj}} & R[\mathtt{efalse}]
\end{array}$$

Note it is possible to compute with open expressions using the above definition.

A few simple properties concerning computation are the following.

LEMMA 3.5 (UNIFORMITY OF EVALUATION) (i) $b_0 = b_1$ if $a \mapsto^1_{\text{inj}} b_0$ and $a \mapsto^1_{\text{inj}} b_1$

(ii) $a \mapsto^1_{\text{inj}} b$ implies $a\,[\sigma] \mapsto^1_{\text{inj}} b\,[\sigma]$

(iii) $a \mapsto^1_{\text{inj}} b$ implies $R[a] \mapsto^1_{\text{inj}} R[b]$

We now define an alternative but equivalent notion of $\sqsubseteq_{\text{inj}}$, restricting the space of contexts to be **c**losed **i**nstances of all **u**ses of an expression. This equivalence is thus called **ciu** equivalence, $\cong^{\text{ciu}}_{\text{inj}}$, following (Mason and Talcott 1991). $a \cong^{\text{ciu}}_{\text{inj}} b$ means a and b behave identically when closed (the *closed instances* part) and placed in any reduction context R (the *uses* part).

DEFINITION 3.6 (CIU ORDERING, $\sqsubseteq^{\text{ciu}}_{\text{inj}}$)

$a \sqsubseteq^{\text{ciu}}_{\text{inj}} b$ iff for all R, σ such that $R[a\,[\sigma]], R[b\,[\sigma]]$ closed, $R[a\,[\sigma]]\downarrow$ implies $R[b\,[\sigma]]\downarrow$

THEOREM 3.7 (CIU) $a \sqsubseteq_{\text{inj}} b$ iff $a \sqsubseteq^{\text{ciu}}_{\text{inj}} b$.

For a proof, see (Mason, Smith, and Talcott 1996). For this simple language it is even possible to characterize $\sqsubseteq_{\text{inj}}$ via a bisimulation ordering (Howe 1996). We now list a collection of basic $\cong_{\text{inj}}$ / $\sqsubseteq_{\text{inj}}$ properties, all easily provable from Theorem 3.7.

LEMMA 3.8 (BASIC $\sqsubseteq_{\text{inj}}$ / $\cong_{\text{inj}}$ PROPERTIES) (i) If $a \sqsubseteq_{\text{inj}} b$, then for $v \in \mathbb{V}^{\emptyset}_{\text{inj}}$, $a\,[v/x] \sqsubseteq_{\text{inj}} b\,[v/x]$.

(ii) $\mathtt{bot} \sqsubseteq_{\text{inj}} a$.

(iii) For closed a, $a\uparrow$ iff $a \cong_{\text{inj}} \mathtt{bot}$.

(iv) $R[\mathtt{bot}] \cong_{\text{inj}} \mathtt{bot}$.

(v) $\cong_{\text{inj}}$ respects computation, i.e. $a \cong_{\text{inj}} b$ if $a \mapsto_{\text{inj}} b$.

(vi) If $a \mapsto_{\text{inj}} \lambda x.b$ and $a \cong_{\text{inj}} a'$ then $a' \cong_{\text{inj}} \lambda x.b'$ for some $b' \cong_{\text{inj}} b$.

(vii) If $a \mapsto_{\text{inj}} \mathtt{inj}_i(b)$ and $a \cong_{\text{inj}} a'$ then $a' \cong_{\text{inj}} \mathtt{inj}_i(b')$ for some $b' \cong_{\text{inj}} b$.

(viii) $a \cong_{\text{inj}} \mathtt{out}_i(\mathtt{inj}_i(a))$.

(ix) If $y \notin \text{FreeVars}(\lambda x.b)$, then $\lambda x.b \cong_{\text{inj}} \lambda y.(\lambda x.b)\,(\,y\,)$.

(x) Extensionality: $\lambda x.a_0 \sqsubseteq_{\sim\text{inj}} \lambda x.a_1$ if and only if $(\lambda x.a_0)\,(\,v\,) \sqsubseteq_{\sim\text{inj}} (\lambda x.a_1)\,(\,v\,)$ for all values v.

Properties of the above general form may be proved across a wide range of languages (Mason and Talcott 1991; Gordon and Rees 1996; Talcott 1989). They may in theory be proved directly by induction on computation length (Talcott 1989), but it is far more effective to prove them via first establishing an alternate characterization of $\sqsubseteq_{\sim\text{inj}}$ via $\sqsubseteq^{\text{ciu}}_{\sim\text{inj}}$ or a bisimulation ordering. For all of the above properties except extensionality, either **ciu** or bisimulation characterizations allow for direct proofs; extensionality is trivial to establish via a bisimulation characterization but requires some work when proved via **ciu**.

As was the case for $\sqsubseteq_{\sim\text{inj}}$, an alternate characterization of $\{\cdot\} \sqsubseteq_{\sim\text{inj}} \{\cdot\}$ is needed to facilitate proofs. An analogue of **ciu** ordering may be defined for $\{\cdot\} \sqsubseteq_{\sim\text{inj}} \{\cdot\}$. Bisimulation characterizations of $\{\cdot\} \sqsubseteq_{\sim\text{inj}} \{\cdot\}$ are also possible.

DEFINITION 3.9 (CIU SET ORDERING $\sqsubseteq^{\text{ciu}}_{\sim\text{inj}}$) $A \sqsubseteq^{\text{ciu}}_{\sim\text{inj}} B$ for $\sqsubseteq_{\sim\text{inj}}$-directed A and B if and only if for all $a \in A$ and for all σ, R such that $R[A\,[\sigma]]$ and $R[B\,[\sigma]]$ are sets of closed expressions, if $R[a\,[\sigma]]\downarrow$ then there exists $b \in B$ such that $R[b\,[\sigma]]\downarrow$.

The main characterization theorem is

THEOREM 3.10 (SET ORDERING CIU) $A \sqsubseteq_{\sim\text{inj}} B$ iff $A \sqsubseteq^{\text{ciu}}_{\sim\text{inj}} B$.

This is Theorem 4.6 of (Mason, Smith, and Talcott 1996).

$\{\cdot\} \sqsubseteq_{\sim\text{inj}} \{\cdot\}$ has the important property of allowing fixed points to be approximated. Fixed points may be shown equivalent to their set of finite unrollings. This breaks the cycle of a fixed point and gives an induction principle for reasoning about recursive functions. We make the following abbreviation: for a functional $f = \lambda x.\lambda y.a$, define $f^0 = \lambda x.\mathtt{bot}$ and $f^{n+1} = f\,(\,f^n\,)$.

LEMMA 3.11 (FIXED POINT) For a functional f,

(i) $\{\mathtt{fix}(\,f\,)\} \cong_{\text{inj}} \{f^n \mid n \in \mathbb{N}\}$,

(ii) $\mathtt{fix}(\,f\,) \cong_{\text{inj}} f\,(\mathtt{fix}(\,f\,))$, and

(iii) for all a, $\lambda x.a \cong_{\text{inj}} f\,(\lambda x.a\,)$ implies $\mathtt{fix}(\,f\,) \sqsubseteq_{\sim\text{inj}} \lambda x.a$.

3.2 Finite Elements

The problem we focus on in this paper is obtaining additional proof principles using the finite algebraic structure of the language. As mentioned in the introduction, this allows reasoning about infinite elements in terms of their finite elements since each infinite element is the lub of all smaller finite elements.

We show in this section how finite elements may be defined in an operational semantics. This material is taken from (Mason, Smith, and Talcott 1996), where complete proofs are also to be found.

With the fixed point property we saw how recursive functions could be decomposed into finite components for inductive reasoning. Similar properties could be proved for other particular structures, such as lists formed via iterative pairing. However, what would be even more desirable would be a general principle for decomposing *all* programs into finite components. The finite decomposition of functions above still does not decompose the argument or return value of the function, which may still be infinite.

The finite expressions accomplish precisely this goal: any expression may be decomposed into a set of finite approximations stratified by level k that are finite in the sense that there are only finitely many distinct approximations up to operational equivalence at any level k. The finite decomposition of expressions is critical to constructions that define self-referential structures (MacQueen, Plotkin, and Sethi 1984; Pitts 1996), for it gives an inductive structure by which self-referentiality may be avoided.

We construct finite expressions "top-down", by syntactically projecting arbitrary expressions to produce finite expressions. This is the opposite of domain construction, which starts with only finite elements. In order for projections to be performed, the presence of recognizer operators $\mathtt{is}\ldots$ in the language is critical: each sort in a multi-sorted language is projected in a different manner, and recognizers allow a run-time projection operation to be defined.

DEFINITION 3.12 (FINITE PROJECTIONS π^n_{inj}) The projection functional π_{inj}, finite projections π^n_{inj}, and infinite projection $\pi^\infty_{\mathrm{inj}}$ are defined as follows.

$$
\begin{aligned}
\pi_{\mathrm{inj}} &= \lambda y.\lambda x. \\
&\qquad \mathtt{eif}(\mathtt{is}_1(x), \mathtt{inj}_1(y(\mathtt{out}_1(x)))), \\
&\qquad \mathtt{eif}(\mathtt{is}_2(x), \mathtt{inj}_2(y(\mathtt{out}_2(x)))), \ldots, \\
&\qquad \mathtt{eif}(\mathtt{is}_k(x), \mathtt{inj}_k(y(\mathtt{out}_k(x)))), \\
&\qquad y \circ x \circ y) \ldots)) \\
\pi^0_{\mathrm{inj}} &= \lambda x.\mathtt{bot} \\
\pi^{n+1}_{\mathrm{inj}} &= \pi_{\mathrm{inj}}(\pi^n_{\mathrm{inj}}) \\
\pi^\infty_{\mathrm{inj}} &= \mathtt{fix}(\pi_{\mathrm{inj}})
\end{aligned}
$$

It is interesting to observe that for $k = 0$, $\mathbb{L}_{\mathrm{inj}(0)}$ is the pure call-by-value λ-calculus, so the above defines projection operations on the pure λ-calculus. This approach

to finite expressions is not found in Barendregt (Barendregt 1984). Barendregt Chapter 14 does review another similar approach which is worth contrasting: Hyland and Wadsworth's labelled λ-calculus. Rather than defining a syntactic projection $\pi^n_{\text{inj}}(e)$, the expression e is labelled with constant n, producing the *labelled λ-term* e^n. evaluation then projects: $(\lambda x.a^{n+1})(b) \to (a\,[b^n/x])^n$. This is identical to how $\pi^{n+1}(\lambda x.e)$ projects the function argument and result both to be at level n. Other researchers have also studied labeled λ-reduction (Egidi, Honsell, and della Rocca 1992). We do not pursue labelled reduction because the addition of labels changes the language and thus changes operational equivalence (see Lemma 4.16 below). Since the projections π^n_{inj} are definable *within* the language, they are guaranteed not to change the underlying equivalence.

Note that for expressions in a simply-typed λ-calculus, there is no need for run-time projection operations: given the type of an expression e, $\pi^k(e)$ can be partially evaluated to remove all cases on the sort, as the type itself reveals the sort This is one way to characterize Milner's construction (Milner 1977).

The following lemma establishes elementary properties of the syntactic projections. For brevity henceforward we drop the subscript inj from the projections π^n.

LEMMA 3.13 (ELEMENTARY π^n/π^∞ PROPERTIES)

$$\begin{array}{rl}
\text{(fix)} & \pi^\infty \cong_{\text{inj}} \{\pi^n \mid n \in \mathbb{N}\} \\
\text{(idemp)} & \pi^n \circ \pi^n \cong_{\text{inj}} \pi^n, \quad \pi^\infty \circ \pi^\infty \cong_{\text{inj}} \pi^\infty \\
\text{(compose)} & \pi^m \circ \pi^n \cong_{\text{inj}} \pi^{\min(m,n)} \\
\text{(order)} & \pi^n \sqsubseteq_{\text{inj}} \pi^{n+1} \sqsubseteq_{\text{inj}} \pi^\infty \\
\text{(inject)} & \pi^{n+1}(\mathtt{inj}_i(v)) \cong_{\text{inj}} \mathtt{inj}_i(\pi^n(v)), \pi^\infty(\mathtt{inj}_i(v)) \cong_{\text{inj}} \mathtt{inj}_i(\pi^\infty(v)) \\
\text{(fun.0)} & \pi^1(\lambda x.e) \cong_{\text{inj}} \lambda x.\mathtt{bot} \\
\text{(fun.+)} & \pi^{n+1}(\lambda x.a) \cong_{\text{inj}} \pi^n \circ \lambda x.a \circ \pi^n, \quad \pi^\infty(\lambda x.a) \cong_{\text{inj}} \pi^\infty \circ \lambda x.a \circ \pi^\infty \\
\text{(prune)} & \pi^n(a) \sqsubseteq_{\text{inj}} a, \quad \pi^\infty(a) \sqsubseteq_{\text{inj}} a \\
\text{(value)} & \pi^\infty(v)\downarrow \text{ for all closed values } v
\end{array}$$

The Finite Approximation Theorem is a key result: any expression is equivalent to its set of finite projections, yielding an inductive decomposition of every expression into its finite counterparts. This property is a close analogue of the ω-algebraicity property of domains.

THEOREM 3.14 (FINITE APPROXIMATION) $\{\pi^n(a) \mid n \in \mathbb{N}\} \cong_{\text{inj}} \{a\}$.

To prove this, we show that for any particular computation, a large enough projection n will suffice. However, a direct proof of this property is a bookkeeping nightmare as different projection values π^k may percolate throughout the expression during evaluation. A simpler proof is to characterize the limit of the projection function. π^∞ may be characterized as an identity function.

LEMMA 3.15 (IDENTITY OF π^∞) $\pi^\infty \cong_{\text{inj}} \lambda x.x$

To prove this, we characterize how the projections π^∞ may percolate throughout expressions during evaluation. Inductively define $\tau(a)$ and $\tau(R)$ as follows:

$$\begin{array}{l}\tau(x) = x \\ \tau(\mathtt{op}^-(a_0, \ldots, a_n)) = \pi^\infty(\mathtt{op}^-(\tau(a_0), \ldots, \tau(a_n))) \\ \tau(\mathtt{inj}_i(a)) = \mathtt{inj}_i(\tau(a)) \\ \tau(\lambda x.a) = \pi^\infty \circ \lambda x.\tau(a) \circ \pi^\infty \\ \tau(R) = \tau(R[x])\,[\bullet/x]\end{array}$$

where operator $\mathtt{op}^-$ is either $\mathtt{app}$, $\mathtt{out}_i$, or $\mathtt{is}_i$. This definition is carefully chosen to have properties corresponding to how π^∞ percolates through evaluation; notice for instance $\tau(v)$ is a value for any value v. The identity of π^∞ may be proved by establishing that $a\downarrow$ implies $\tau(a)\downarrow$; this then establishes the Finite Approximation Theorem. In Section 5.2 below, analogous results are established for a language with state; more complete proofs are given there.

The syntactic projections may rightfully be called "finite expressions": all expressions that are finite in the classical sense of Definition 2.7 are equivalent to some syntactically projected expression, and the cardinality of each level is finite.

THEOREM 3.16 (FINITENESS CHARACTERIZATION) (i) For all $n \in \mathbb{N}$,

$$\{a \mid a \in \mathbb{E}^\emptyset_{\text{inj}} \text{ and } a \cong_{\text{inj}} \pi^n(a)\}$$

contains finitely many $\cong_{\text{inj}}$-distinct expressions.

(ii) For all $a \in \mathbb{E}^\emptyset_{\text{inj}}$ and $n \in \mathbb{N}$, $\pi^n(a)$ is finite in the sense of Definition 2.7.

(iii) $\mathbb{E}^\omega_{\text{inj}} = \bigcup_{n\in\mathbb{N}}\{a \mid a \in \mathbb{E}^\emptyset_{\text{inj}} \text{ and } a \cong_{\text{inj}} \pi^n(a)\}$

Furthermore, $\{\cdot\} \sqsubseteq_{\text{inj}} \{\cdot\}$-directed sets all may be shown to have least upper bounds, and $\sqsubseteq_{\text{inj}}$ is ω-algebraic; see (Mason, Smith, and Talcott 1996) for these properties and a proof of the previous Theorem.

4 A Space Of Languages

In the previous section a very simple language family $\mathbb{L}_{\text{inj}(k)}$ was studied. In this section we study progressively more complex languages and consider what results still hold from the development for $\mathbb{L}_{\text{inj}(k)}$, and what results fail. High-level languages $\mathbb{L}_h$ are studied by defining embeddings of $\mathbb{L}_h$ into the injection languages $\mathbb{L}_{\text{inj}(k)}$. This allows for us to quickly present a sequence of languages, and also helps highlight how small (or large) the gap is between different languages.

Notions of operational ordering $\sqsubseteq$ and operational set ordering $\{\cdot\} \sqsubseteq \{\cdot\}$ were defined on arbitrary operational structures in Section 2, so there is no question of the generality of those definitions. Our main goal here is to study how well the finite elements may be characterized, and in particular if Theorems 3.14 and 3.16 stated above for $\mathbb{L}_{\text{inj}(k)}$ may be proved for these more complex languages.

4.1 Booleans, Pairs, and Numbers

We now define a language $\mathbb{L}_{\mathrm{bpn}(k)}$ which adds booleans, pairs, and numbers to $\mathbb{L}_{\mathrm{inj}(k)}$. We preserve the $\mathtt{inj}_k/\mathtt{out}_k/\mathtt{is}_k$ operators to allow this language to be extended in turn. The case $k = 0$ yields a language with no injections, and $k = 2$ yields the standard left/right injections.

DEFINITION 4.1 $\mathbb{L}_{\mathrm{bpn}(k)}$ has structure $\langle \mathbb{E}, \mathbb{V}, \mathbb{O}, \mapsto \rangle$ where

$$\begin{array}{rcl} \mathbb{O} & = & \mathbb{O}_{\mathrm{inj}(k)} \cup \{\mathtt{true}, \mathtt{false}, \mathtt{isbool}(e), 0, 1, 2, \ldots, \mathtt{succ}(e), \mathtt{pred}(e), \\ & & \mathtt{iszero}(e), \mathtt{isnat}(e), \mathtt{pr}(e, e'), \mathtt{fst}(e), \mathtt{snd}(e), \mathtt{ispr}(e), \mathtt{if}(e, e', e'')\} \end{array}$$

We include recognizer operators $\mathtt{isbool}$, $\mathtt{isnat}$, $\mathtt{ispr}$ in $\mathbb{L}_{\mathrm{bpn}(k)}$. In Section 4.2 below we consider the alternative case when there are no recognizers. We will very briefly outline the evaluator for $\mathbb{L}_{\mathrm{bpn}(k)}$ (and, will not define explicit evaluators for most of the languages that follow). For brevity we will define some notions as extensions of the notions defined in the presentation of $\mathbb{L}_{\mathrm{inj}(k)}$ of Section 3, reading those definitions with $\mathbb{E}_{\mathrm{bpn}(k)}$ in place of $\mathbb{E}_{\mathrm{inj}(k)}$. The values $\mathbb{V}_{\mathrm{bpn}(k)}$ include the cases as $\mathbb{L}_{\mathrm{inj}(k)}$ plus numbers, booleans, and $\mathtt{pr}(v, v)$. The reduction contexts $\mathbb{R}_{\mathrm{bpn}(k)}$ include the same cases plus $\mathtt{if}(R[\bullet], e, e)$, $\mathtt{pred}(R[\bullet])$, $\mathtt{succ}(R[\bullet])$, $\mathtt{iszero}(R[\bullet])$, $\mathtt{fst}(R[\bullet])$, $\mathtt{snd}(R[\bullet])$, $\mathtt{pr}(R[\bullet], e)$, $\mathtt{pr}(v, R[\bullet])$, $\mathtt{isbool}(R[\bullet])$, and $\mathtt{isnat}(R[\bullet])$ and $\mathtt{ispr}(R[\bullet])$.

DEFINITION 4.2 ($\mapsto^1_{\mathrm{bpn}(k)}$, $\mapsto_{\mathrm{bpn}(k)}$) $\mapsto_{\mathrm{bpn}(k)}$ is the transitive, reflexive closure of $\mapsto^1_{\mathrm{bpn}(k)}$:

$$\begin{array}{rcl} R[e] & \mapsto^1_{\mathrm{bpn}(k)} & R[e'] \\ & \text{where } R[e] \mapsto^1_{\mathrm{inj}(k)} R[e'] \text{ is a case of Definition 3.4} & \\ R[\mathtt{if}(\mathtt{true}, a, b)] & \mapsto^1_{\mathrm{bpn}(k)} & R[a] \text{ and } \mapsto^1_{\mathrm{bpn}(k)} R[b] \text{ for } \mathtt{false} \text{ case} \\ R[\mathtt{fst}(\mathtt{pr}(v, v'))] & \mapsto^1_{\mathrm{bpn}(k)} & R[v] \text{ and } \mapsto^1 R[v'] \text{ for } \mathtt{snd} \\ R[\mathtt{succ}(v)] & \mapsto^1_{\mathrm{bpn}(k)} & R[v+1] \text{ for } v \in \mathbb{N} \\ R[\mathtt{pred}(v+1)] & \mapsto^1_{\mathrm{bpn}(k)} & R[v] \text{ for } v \in \mathbb{N} \\ R[\mathtt{iszero}(v)] & \mapsto^1_{\mathrm{bpn}(k)} & R[b] \\ & \text{for } v \in \mathbb{N}, b \text{ a boolean, and } b = \mathtt{true} \text{ iff } v = 0 & \\ R[\mathtt{is}\{\mathtt{bool}/\mathtt{pr}/\mathtt{num}\}(v)] & \mapsto^1_{\mathrm{bpn}(k)} & R[b] \\ & \text{where } v \notin \mathbb{X} \text{ and boolean } b \text{ is } \mathtt{true} \text{ iff } v \text{ is a } \mathtt{bool}/\mathtt{pr}/\mathtt{num} & \end{array}$$

$\mathbb{L}_{\mathrm{bpn}(k)}$ is mapped to low-level language $\mathbb{L}_{\mathrm{inj}(k+3)}$ by the following homomorphic embedding.

DEFINITION 4.3 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(k)} \to \mathbb{L}_{\mathrm{inj}(k+3)}$ is defined as $C_{\mathrm{init}} = \bullet$ and an embedding of expressions as follows.

$$
\begin{array}{rcl}
[\![\mathtt{true}]\!] & = & \mathtt{inj}_{k+3}(\mathtt{etrue}) \\
[\![\mathtt{false}]\!] & = & \mathtt{inj}_{k+3}(\mathtt{efalse}) \\
[\![\mathtt{if}(e,e',e'')]\!] & = & \mathtt{out}_{k+3}([\![e]\!])\,(\lambda x.[\![e']\!])\,(\lambda x.[\![e'']\!]) \text{ for fresh } x \\
[\![\mathtt{isbool}(a)]\!] & = & \mathtt{eif}(\mathtt{is}_{k+3}([\![a]\!]),[\![\mathtt{true}]\!],[\![\mathtt{false}]\!]) \\
[\![\mathtt{pr}(a,b)]\!] & = & \mathtt{inj}_{k+2}((\lambda x.\lambda y.\lambda p.p\,(x)\,(y))\,([\![a]\!])\,([\![b]\!])) \\
[\![\mathtt{fst}(a)]\!] & = & \mathtt{out}_{k+2}([\![a]\!])\,(\lambda x.\lambda y.x) \\
[\![\mathtt{snd}(a)]\!] & = & \mathtt{out}_{k+2}([\![a]\!])\,(\lambda x.\lambda y.y) \\
[\![\mathtt{ispr}(a)]\!] & = & \mathtt{eif}(\mathtt{is}_{k+2}([\![a]\!]),[\![\mathtt{true}]\!],[\![\mathtt{false}]\!]) \\
[\![n]\!] & = & \mathtt{inj}_{k+1}([\![\overbrace{\mathtt{pr}(\mathtt{true},\ldots\mathtt{pr}(\mathtt{true}}^{n},\mathtt{pr}(\mathtt{false},\mathtt{false}))\ldots)]\!]) \\
[\![\mathtt{succ}(e)]\!] & = & \mathtt{let}\ x = \mathtt{out}_{k+1}([\![e]\!])\ \mathtt{in}\ \mathtt{inj}_{k+1}([\![\mathtt{pr}(\mathtt{true},x)]\!]) \\
[\![\mathtt{pred}(e)]\!] & = & \mathtt{let}\ x = \mathtt{out}_{k+1}([\![e]\!])\ \mathtt{in}\ \mathtt{inj}_{k+1}(\mathtt{inj}_{k+2}(\mathtt{out}_{k+2}([\![\mathtt{snd}(x)]\!])))) \\
[\![\mathtt{iszero}(e)]\!] & = & \mathtt{let}\ x = \mathtt{out}_{k+1}([\![e]\!])\ \mathtt{in} \\
 & & \quad \mathtt{eif}(\mathtt{out}_{k+3}([\![\mathtt{fst}(x)]\!]),[\![\mathtt{false}]\!],[\![\mathtt{true}]\!]) \\
[\![\mathtt{isnat}(a)]\!] & = & \mathtt{eif}(\mathtt{is}_{k+1}([\![a]\!]),[\![\mathtt{true}]\!],[\![\mathtt{false}]\!]) \\
[\![e]\!] & = & \text{homomorphic for all other } e \in \mathbb{E}_{\mathrm{bpn}(k)}
\end{array}
$$

The mapping for contexts extends the above with the case $[\![\bullet]\!] = \bullet$. Observe how operator $\mathtt{inj}_{k+3}$ wraps booleans, $\mathtt{inj}_{k+2}$ wraps pairs, and $\mathtt{inj}_{k+1}$ wraps numerals, keeping these datatypes disjoint. Boolean and pair encodings are the classic Church encodings; we then use the booleans and pairs to encode numerals.

LEMMA 4.4 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(k)} \to \mathbb{L}_{\mathrm{inj}(k+3)}$ is a sound embedding homomorphic in $\mathbb{O}_{\mathrm{inj}(k)}$.

PROOF: Show $a \downarrow_{\mathrm{bpn}(k)}$ iff $[\![a]\!] \downarrow_{\mathrm{inj}(k+3)}$. We may show $a \mapsto^1_{\mathrm{bpn}(k)} b$ implies $[\![a]\!] \cong_{\mathrm{inj}(k+3)} [\![b]\!]$ by inspection of the cases of $\mapsto^1_{\mathrm{inj}(k+3)}$. Also, we may show that if $a \not\mapsto^1_{\mathrm{bpn}(k)}$, *i.e.,* it is stuck, then $[\![a]\!] \not\downarrow_{\mathrm{inj}(k+3)}$. And, from the fact that for all values $v \in \mathbb{V}_{\mathrm{bpn}(k)}$, $[\![v]\!] \downarrow_{\mathrm{inj}(k+3)}$, the result follows.

The embedding can easily be seen to be homomorphic in $\mathbb{O}_{\mathrm{inj}(k)}$. □

Additional abbreviations for $\mathbb{E}_{\mathrm{bpn}(k)}$ include

$$
\begin{array}{rcl}
\mathtt{nateq}(e,e') & = & \mathtt{fix}(\lambda f.\lambda x.\lambda y.\mathtt{if}(\mathtt{iszero}(x),\mathtt{iszero}(y), \\
 & & \quad \mathtt{if}(\mathtt{iszero}(y),\mathtt{false},f\,(\mathtt{pred}(x))\,(\mathtt{pred}(y))))\,(e)\,(e') \\
\mathtt{let}\ \mathtt{pr}(x_1,x_2) = e\ \mathtt{in}\ e' & = & \mathtt{let}\ x = e\ \mathtt{in} \\
 & & \quad \mathtt{let}\ x_1 = \mathtt{fst}(x)\ \mathtt{in} \\
 & & \quad\quad \mathtt{let}\ x_2 = \mathtt{snd}(x)\ \mathtt{in}\ e', \text{ for } x \text{ fresh}
\end{array}
$$

If the embedding did not use injections, numbers and functions would be of the same sort, and $\mathbb{L}_{\mathrm{bpn}}$ computations such as $0\,(\lambda x.x)$ would terminate when they should be stuck. This fact is expressed in the following Lemma, which shows the pure λ-calculus is too weak to serve as an embedding target.

LEMMA 4.5 There is no sound embedding $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(0)} \to \mathbb{L}_{\mathrm{inj}(0)}$ homomorphic in $\mathbb{O}_{\mathrm{inj}(0)}$.

PROOF: Suppose there was. $[\![0]\!] \cong_{\mathrm{inj}(0)} \lambda x.e_0$ and $[\![1]\!] \cong_{\mathrm{inj}(0)} \lambda x.e_1$ since $[\![0]\!]\downarrow_{\mathrm{inj}(0)}$ and $[\![1]\!]\downarrow_{\mathrm{inj}(0)}$. Consider $[\![\mathtt{iszero}(e)]\!]$; since $[\![\mathtt{true}]\!]$ and $[\![\mathtt{false}]\!]$ must be non-equivalent, and $[\![0]\!]$ and $[\![1]\!]$ must be non-equivalent, this $\mathbb{L}_{\mathrm{inj}}$ computation must touch expression e, so for e being $[\![0]\!]$ it will compute to $R[[\![0]\!](\lambda x.e')]$ for some $\lambda x.e'$ (this being the first place where $[\![0]\!]$ is touched), and go on to terminate. Thus $[\![0]\!](\lambda x.e')\downarrow_{\mathrm{inj}(0)}$. Since the embedding is homomorphic, $[\![\lambda x.e']\!] = \lambda x.e'$, and so $[\![0(\lambda x.e')]\!]\downarrow_{\mathrm{inj}(0)}$, a contradiction. □

LEMMA 4.6 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(k)} \to \mathbb{L}_{\mathrm{inj}(k+3)}$ is not fully abstract.

PROOF: $\lambda x.\mathtt{if}(\mathtt{ispr}(x), (\lambda x.0)(\mathtt{fst}(x)), 0) \cong_{\mathrm{bpn}(k)} \lambda x.0$ but the $\mathbb{L}_{\mathrm{inj}(k+3)}$ context $(\bullet)(\mathtt{inj}_{k+2}(\lambda x.\mathtt{bot}))$ distinguishes the embedded forms. □

Still, since the embedding is homomorphic, equations from $\mathbb{L}_{\mathrm{inj}(k+3)}$ can be directly lifted to $\mathbb{L}_{\mathrm{bpn}(k)}$. One concrete example is the fixed point lemma.

LEMMA 4.7 $\{\mathtt{fix}(\lambda x.\lambda z.f(x)(z))\} \cong_{\mathrm{bpn}(k)} \{\lambda x.\lambda z.f^n(x)(z) \mid n \in \mathbb{N}\}$.

PROOF: This property is proved for $\mathbb{L}_{\mathrm{inj}(k)}$ as Lemma 3.11, and since the expressions only involve operators homomorphic in the embedding $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(k)} \to \mathbb{L}_{\mathrm{inj}(k+3)}$ (taking f to be a variable and not a metavariable), the result follows by Lemma 2.13. □

The finite approximation theorem (3.14) can also be "lifted" from $\mathbb{L}_{\mathrm{inj}}$, but the syntactic projection functions π will not project pairs or numbers and will thus be incomplete; in particular 3.16 will not be provable. To obtain complete projection functions, fuller forms may be directly defined in $\mathbb{L}_{\mathrm{bpn}(k)}$.

DEFINITION 4.8 (FINITE PROJECTIONS π^n_{bpn})

$$
\begin{aligned}
\pi_{\mathrm{bpn}} &= \lambda y.\lambda x.\\
&\qquad \mathtt{eif}(\mathtt{is}_1(x), \mathtt{inj}_1(y(\mathtt{out}_1(x)))),\\
&\qquad \mathtt{eif}(\mathtt{is}_2(x), \mathtt{inj}_2(y(\mathtt{out}_2(x)))), \ldots,\\
&\qquad \mathtt{eif}(\mathtt{is}_k(x), \mathtt{inj}_k(y(\mathtt{out}_k(x)))),\\
&\qquad \mathtt{if}(\mathtt{isbool}(x), x,\\
&\qquad \mathtt{if}(\mathtt{ispr}(x), \mathtt{pr}(y(\mathtt{fst}(x)), y(\mathtt{snd}(x)))),\\
&\qquad \mathtt{if}(\mathtt{isnat}(x), \mathtt{if}(\mathtt{iszero}(x), 0, \mathtt{succ}(y(\mathtt{pred}(x))))),\\
&\qquad y \circ x \circ y)) \ldots))\\
\pi^0_{\mathrm{bpn}} &= \lambda x.\mathtt{bot}\\
\pi^{n+1}_{\mathrm{bpn}} &= \pi_{\mathrm{bpn}(k)}(\pi^n_{\mathrm{bpn}(k)})\\
\pi^{\infty}_{\mathrm{bpn}(k)} &= \mathtt{fix}(\pi_{\mathrm{bpn}})
\end{aligned}
$$

These finite projections in turn allow the finite elements of $\mathbb{L}_{\mathrm{bpn}}$ to be characterized.

THEOREM 4.9 Theorems 3.14 and 3.16 hold for $\mathbb{L}_{\mathrm{bpn}(k)}$ with projections π^n_{bpn}.

Proofs of these theorems are found in (Mason, Smith, and Talcott 1996). The principle of induction on the finite rank n is thus successfully obtained for this language.

4.2 Booleans, pairs, and numbers without recognizers

It is interesting to consider ramifications of languages without recognizers `isnat`, `ispr`, `isbool`, and $\mathtt{is}_k$. In such languages the case analysis programmed into the syntactic projection functions π^n_{bpn} cannot be programmed, and thus the approach of Section 3.2 is not directly possible. We show there still is some capability of reasoning using finite projections, but at the cost of fully abstract reasoning.

DEFINITION 4.10 (i) $\mathbb{L}_{\mathrm{inj}-(k)}$ is defined to be $\mathbb{L}_{\mathrm{inj}(k)}$ with recognizer operators $\mathtt{is}_i$, $1 \leq i \leq k$, removed from $\mathbb{O}_{\mathrm{inj}(k)}$ and the $\mathtt{is}_i$ cases removed from $\mapsto_{\mathrm{inj}(k)}$.

(ii) $\mathbb{L}_{\mathrm{bpn}-(k)}$ is defined to be $\mathbb{L}_{\mathrm{bpn}(k)}$ without recognizer operators `isnat`, `ispr`, `isbool`, or $\mathtt{is}_k$ for any k.

(iii) $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}-(k)} \to \mathbb{L}_{\mathrm{inj}-(k+3)}$ is then the obvious restriction of the above embedding (Definition 4.3) that removes the recognizer cases. (Notice that $\mathtt{is}_k$ is not used in any non-recognizer cases).

This embedding is getting closer to being fully abstract than the mapping with recognizers, but there are some subtle cases where parametricity is observable in $\mathbb{L}_{\mathrm{inj}-(k)}$ but masked in $\mathbb{L}_{\mathrm{bpn}-(k)}$.

LEMMA 4.11 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}-(k)} \to \mathbb{L}_{\mathrm{inj}-(k+3)}$ is not fully abstract.

PROOF:

$$\lambda x.\mathtt{let}\ y = \mathtt{fst}(x)\ \mathtt{in}\ \mathtt{let}\ z = \mathtt{snd}(x)\ \mathtt{in}\ x \cong_{\mathrm{bpn}-(k)}$$
$$\lambda x.\mathtt{let}\ y = \mathtt{fst}(x)\ \mathtt{in}\ \mathtt{let}\ z = \mathtt{snd}(x)\ \mathtt{in}\ \mathtt{pr}(y,z)$$

but the $\mathbb{L}_{\mathrm{inj}-(k+3)}$ context $\mathtt{out}_{k+2}((\bullet)(\mathtt{inj}_{k+2}(\lambda p.p(\lambda x.\mathtt{bot}))))(\lambda x.x)(\lambda x.x)$ distinguishes the embedded forms. □

A similar problem arises with booleans, so neither pairing nor booleans and conditional can be encoded in a full and faithful manner. Numbers however can probably be encoded faithfully in a language with only pairs and booleans.

We now establish that removal of recognizers causes the equivalence to change. This means we are not completely free to add or remove recognizers from languages as tools (*e.g.,* to use them to define the finite projections), because the underlying equivalence will be altered and full abstraction will then fail.

LEMMA 4.12 For all g, if $\mathtt{iszero}(g(0))\downarrow_{\mathrm{bpn}-(k)}$ and $\mathtt{if}(g(\mathtt{true}),0,0)\downarrow_{\mathrm{bpn}-(k)}$, then $g \cong \lambda x.x$.

PROOF: First observe that this requires $g(0/\mathtt{true})\downarrow_{\mathrm{bpn}-(k)}$. g must then never touch its argument, for suppose it did: then g must compute to $R[0/\mathtt{true}]$ for some R. Since $\mathbb{E}_{\mathrm{bpn}-(k)}$ contains no recognizers, this state will get stuck on at least one of 0 or $\mathtt{true}$ filling the hole (note that $\mathtt{iszero}(\mathtt{true})$ is stuck), contradiction. So, since g does not touch its argument, the result of $g(0/\mathtt{true})$ must be a parametric value of the form $C[0/\mathtt{true}]$. The only case then for which both $\mathtt{iszero}(C[0])$ and $\mathtt{if}(C[\mathtt{true}],0,0)$ do not get stuck is then when $C[x]\mapsto_{\mathrm{bpn}-(k)} x$. Thus, $g(v)\cong v$ for any v, and the result follows by extensionality. □

LEMMA 4.13 $\sqsubseteq_{\sim\mathrm{bpn}(k)}\subsetneq\sqsubseteq_{\sim\mathrm{bpn}-(k)}$ when these relations are restricted to expressions of $\mathbb{E}_{\mathrm{bpn}-(k)}$.

PROOF: Consider the $\mathbb{L}_{\mathrm{bpn}-(k)}$ context

$$C \;=\; \lambda z.\mathtt{if}(z(\mathtt{true}),\mathtt{if}(\mathtt{nateq}(z(0),0),\bullet,0),0)$$

Then, $C[\mathtt{pr}(0,0)]\sqsubseteq_{\sim\mathrm{bpn}-(k)} C[z(\mathtt{pr}(0,0))]$: if $C[\mathtt{pr}(0,0)](g)\downarrow_{\mathrm{bpn}-(k)}$, then by Lemma 4.12, $g\cong_{\mathrm{bpn}-(k)}\lambda x.x$, and so $C[z(\mathtt{pr}(0,0))](g)\downarrow_{\mathrm{bpn}-(k)}$ as well. However, the $\mathbb{L}_{\mathrm{bpn}(k)}$ context $\bullet(\lambda x.\mathtt{if}(\mathtt{ispr}(x,\mathtt{bot},x)))$ distinguishes the two expressions. □

4.2.1 Finite Projections Without Recognizers

Since $\mathbb{L}_{\mathrm{bpn}-(k)}$ contains no recognizers, the syntactic projection operations π^n_{bpn} are not definable within the language as was possible in the presence of recognizers. Without some other approach, no finite element structure can be developed for such a language.

It is possible to use $\sqsubseteq_{\sim\mathrm{bpn}(k)}$ to directly prove instances of $\sqsubseteq_{\sim\mathrm{bpn}-(k)}$ via Lemma 4.13, and in the former language the syntactic projection functions may be directly expressed. However, ordering $\sqsubseteq_{\sim\mathrm{bpn}(k)}$ is far too fine-grained, as Lemma 4.13 shows.

One solution is to add the projections π^n_{bpn} of $\mathbb{L}_{\mathrm{bpn}(k)}$ to $\mathbb{L}_{\mathrm{bpn}-(k)}$ as primitive operations $\mathtt{proj}^n$. This means we will not allow all recognizers in testing contexts of $\mathbb{L}_{\mathrm{bpn}-(k)}$; they will only be used in the restricted way the projections use them. We define this language in abbreviated form via a mapping to $\mathbb{L}_{\mathrm{bpn}(k)}$.

DEFINITION 4.14 (i) $\mathbb{E}_{\mathrm{bpn}\pi-(k)}$ is $\mathbb{E}_{\mathrm{bpn}-(k)}$ with additional 0-ary operators $\mathtt{proj}^n$ for each $n\in\mathbb{N}$.

(ii) $[\![\cdot]\!]\in\mathbb{L}_{\mathrm{bpn}\pi-(k)}\to\mathbb{L}_{\mathrm{bpn}(k)}$ is homomorphic in all operators $\mathtt{op}\in\mathbb{O}_{\mathrm{bpn}\pi-(k)}$ except

$$[\![\mathtt{proj}^n]\!] \;=\; \pi^n_{\mathrm{bpn}}$$

THEOREM 4.15 Finite Approximation, Theorem 3.14, is provable for the projections $\mathtt{proj}^n(e)$ of $\mathbb{L}_{\mathrm{bpn}\pi-(k)}$: $\{\mathtt{proj}^n(a)\mid n\in\mathbb{N}\}\cong_{\mathrm{bpn}\pi-(k)}\{a\}$.

PROOF: The results from Theorem 4.9 may directly be applied to $\mathbb{L}_{\mathrm{bpn}\pi-}$ by Lemma 2.9. □

Since $\mathbb{L}_{\mathrm{bpn}\pi-(k)}$ in addition contains operators $\mathtt{proj}^n$, $\precsim_{\mathrm{bpn}\pi-(k)}$ restricted to expressions of $\mathbb{L}_{\mathrm{bpn}-(k)}$ has more testing contexts than $\precsim_{\mathrm{bpn}-(k)}$. These extra contexts can distinguish more expressions, so adding atomic projection operations has an undesirable effect on equivalence.

LEMMA 4.16 $\precsim_{\mathrm{bpn}\pi-(k)} \subsetneq \precsim_{\mathrm{bpn}-(k)}$ when the relations are restricted to $\mathbb{E}_{\mathrm{bpn}-(k)}$.

PROOF: $\subset$ follows from the fact that $\mathbb{E}_{\mathrm{bpn}-(k)} \subseteq \mathbb{E}_{\mathrm{bpn}\pi-(k)}$. They are unequal by the following. The proof of Lemma 4.13 defines a C such that $C[\mathtt{pr}(0,0)] \precsim_{\mathrm{bpn}-(k)} C[z(\mathtt{pr}(0,0))]$. But $C[\mathtt{pr}(0,0)] \not\precsim_{\mathrm{bpn}\pi-(k)} C[z(\mathtt{pr}(0,0))]$, because the context $\bullet(\mathtt{proj}^1)$ distinguishes the two: $C[\mathtt{pr}(0,0)](\mathtt{proj}^1)$ converges, and

$$C[z(\mathtt{pr}(0,0))](\mathtt{proj}^1)$$

diverges because $\mathtt{proj}^1(\mathtt{pr}(0,0))$ diverges. □

A similar problem would likely arise if an explicit expression labeling scheme were attempted in analogy with labeled λ-expressions (Barendregt 1984; Egidi, Honsell, and della Rocca 1992): presence of the labels will cause operational equivalence to change. This shows it is very difficult to characterize the finite structure of untyped languages in a fully abstract manner if the language does not already have recognizer operators as primitives.

4.3 Simple Objects

We briefly study a language with simple functional objects. Simple objects may be defined via a homomorphic embedding. We conjecture here that they may be embedded in $\mathbb{L}_{\mathrm{bpn}}$ in a fully abstract manner.

Our simple objects contain methods that may refer to the object itself. Since they are functional, they do not contain mutable instances, and there is also no notion of class or of method override. Classes (Eifrig, Smith, Trifonov, and Zwarico 1995) and method override (Abadi, Cardelli, and Viswanathan 1996) may also be interpreted via homomorphic embeddings; for brevity we leave them out of this presentation.

DEFINITION 4.17 $\mathbb{L}_{\mathrm{obj}(k)}$ has structure $\langle \mathbb{E}_{\mathrm{obj}(k)}, \mathbb{V}_{\mathrm{obj}(k)}, \mathbb{O}_{\mathrm{obj}(k)}, \mapsto_{\mathrm{obj}(k)} \rangle$ where

$$\begin{aligned} \mathbb{O}_{\mathrm{obj}(k)} \;=\; & \mathbb{O}_{\mathrm{bpn}(k)} \cup \{\mathtt{isobj}(e)\} \cup \{\mathtt{send}_m(e) \mid m \in \mathbb{M}\} \\ & \cup \{\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1, \ldots, x_n.e_n) \mid m_1, \ldots, m_n \in \mathbb{M}\} \end{aligned}$$

for some countable set of messages $\mathbb{M}$; $count(m)$ for $m \in \mathbb{M}$ is a bijection from $\mathbb{M}$ to $\mathbb{N}$.

This notation for objects uses a different operator for each object message form to allow objects to fit the operator arity syntax convention. $\mathbb{L}_{\mathrm{obj}(k)}$ is mapped to lower-level language $\mathbb{L}_{\mathrm{bpn}(k+1)}$ by the following embedding homomorphic in $\mathbb{O}_{\mathrm{bpn}(k)}$ (a composition with $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{bpn}(k)} \to \mathbb{L}_{\mathrm{inj}(k+3)}$ would then yield an embedding into $\mathbb{L}_{\mathrm{inj}(k+4)}$).

DEFINITION 4.18 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ is defined as $C_{\mathrm{init}} = \bullet$ and an embedding of expressions as follows.

$$\begin{array}{rcl}
[\![\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1,\ldots,x_n.e_n)]\!] & = & \mathtt{inj}_{k+1}(\mathtt{fix}(\lambda ob.\lambda y.\\
& & \mathtt{if}(\mathtt{nateq}(y, count(m_1)), (\lambda x.[\![e_1]\!])(ob),\\
& & \mathtt{if}(\ldots,\\
& & \mathtt{if}(\mathtt{nateq}(y, count(m_n)), (\lambda x.[\![e_n]\!])(ob),\\
& & \mathtt{bot})\ldots))))\\
[\![\mathtt{send}_m(e)]\!] & = & \mathtt{out}_{k+1}([\![e]\!])(count(m))\\
[\![\mathtt{isobj}(e)]\!] & = & \mathtt{eif}(\mathtt{is}_{k+1}([\![e]\!]), \mathtt{true}, \mathtt{false})\\
[\![e]\!] & = & \text{homomorphic for all other } e
\end{array}$$

This encoding does not expose the self ($x_1, \ldots, x_n$) since a fixed point is taken (Kamin and Reddy 1994). We conjecture this encoding is fully abstract.

CONJECTURE 4.19 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ is set fully abstract.

If the encoding were not fully abstract, there would be two $\mathbb{L}_{\mathrm{obj}(k)}$ expression sets $A \sqsubseteq_{\mathrm{obj}(k)} B$ but a $\mathbb{L}_{\mathrm{bpn}(k+1)}$ context could distinguish $[\![A]\!]$ and $[\![B]\!]$. However, $\mathbb{L}_{\mathrm{bpn}(k+1)}$ contexts cannot access internal details of object implementations given the representation used. Thus the conjecture is that functional objects are not very complex additions to a language. Establishing this conjecture appears difficult.

A fixed-point principle for reasoning about objects may be easily derived by mapping objects into $\mathbb{L}_{\mathrm{bpn}(k+1)}$ via the above embedding, and lifting the fixed point principle of that language.

DEFINITION 4.20 Given fixed object $\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1,\ldots,x_n.e_n)$, define $\mathtt{obj}^j$ inductively as

$$\begin{array}{l}
\mathtt{obj}^0 = \mathtt{obj}_{m_1,\ldots,m_n}(x_1.\mathtt{bot},\ldots,x_n.\mathtt{bot})\\
\mathtt{obj}^{j+1} = \mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1\,[\mathtt{obj}^j/x_1],\ldots,x_n.e_n\,[\mathtt{obj}^j/x_n])
\end{array}$$

LEMMA 4.21 Given object $\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1,\ldots,x_n.e_n)$,

$$\{\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1,\ldots,x_n.e_n)\} \cong_{\mathrm{obj}(k)} \{\mathtt{obj}^j \mid j \in \mathbb{N}\}.$$

PROOF: It suffices to show

$$[\![\{\mathtt{obj}_{m_1,\ldots,m_n}(x_1.e_1,\ldots,x_n.e_n)\}]\!] \cong_{\mathrm{bpn}(k+1)} [\![\{\mathtt{obj}^j \mid j \in \mathbb{N}\}]\!],$$

and this follows from Lemmas 4.7 and 2.13. □

The simple objects under study are little more than mutually recursive function definitions, so this principle is not much of a generalization over the fixed point property for functions. But, this lifting approach should apply to mappings of more general notions of object.

4.3.1 Finite Projections for Objects

Consider how the syntactic projection operations for objects might be defined, in analogy with the syntactic projections for $\mathbb{L}_{\mathrm{inj}(k)}$ and $\mathbb{L}_{\mathrm{bpn}(k)}$.

$$\begin{aligned}\pi_{\mathrm{obj}} = \lambda y.\lambda x.\\ &\mathtt{if}(\mathtt{isbool}(x),\ldots,\\ &\mathtt{if}(\mathtt{isobj}(x),???,\ldots))\end{aligned}$$

At the "???" point, each method of the object x must be projected, but it is not possible inside $\mathbb{L}_{\mathrm{obj}(k)}$ to detect which methods an arbitrary object has at run-time, so it appears the object projection operation cannot be syntactically expressed. We could at this point fruitfully pursue an alternate theory of objects in which message names had enough of a first-class status so that the projections could be defined. It could indeed be argued that the "proper" notion of an untyped object would allow for first-class message operations (an assertion supported by their presence in the Smalltalk language). But, we elect instead to continue with the current object constructs.

4.4 Simple Objects With Atomic Projections

An alternative is to proceed as we did for $\mathbb{L}_{\mathrm{bpn}-(k)}$ in Section 4.2: extend the language with atomic projection operations $\mathtt{proj}^n_{\mathrm{obj}}$ at the expense of possibly changing the operational equivalence.

DEFINITION 4.22 (i) $\mathbb{O}_{\mathrm{obj}\pi(k)}$ is $\mathbb{O}_{\mathrm{obj}(k)}$ with additional 0-ary operators $\mathtt{proj}^n_{\mathrm{obj}}$ added for each $n \in \mathbb{N}$.

(ii) $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}\pi(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ extends $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ of Definition 4.18 by adding the following clause

$$[\![\mathtt{proj}^n_{\mathrm{obj}}]\!] \;=\; \pi^n_{\mathrm{bpn}}$$

THEOREM 4.23 Finite Approximation, Theorem 3.14, is provable for the projections $\mathtt{proj}^n(e)$ of $\mathbb{L}_{\mathrm{obj}\pi}$: $\{\mathtt{proj}^n(a) \mid n \in \mathbb{N}\} \cong_{\mathrm{obj}\pi} \{a\}$.

PROOF: The results from Theorem 4.9 may directly be applied to $\mathbb{L}_{\mathrm{obj}\pi}$ by Lemma 2.9. □

If the embedding $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ is set fully abstract as conjectured, addition of these projections would fortunately *not* change the underlying equivalence: $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{obj}\pi(k)} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ would also clearly be set fully abstract. A full

and faithful theory of finite objects could then be developed. Thus, there is some potential for developing a full and faithful finite expression theory for objects via this route.

More serious difficulties with finite expressions arise in the presence of effects. We will consider memory effects in the next section.

5 Global effects: Memories

In this section we study a simple memory-based language, $\mathbb{L}_{\mathrm{m}(k)}$. The languages studied in the previous section all could be defined by homomorphic embeddings, because there were no global effects. The addition of global effects thus represents a significant change. We first present an embedding into $\mathbb{L}_{\mathrm{bpn}(k+1)}$. Strachey's memory-threading transformation is used (Milne and Strachey 1976). After establishing some results about this embedding, we define an equivalent direct operational semantics: proving facts via the embedding is cumbersome. We then develop a series of results which characterize the finite elements for a memory-based language.

For simplicity we will define $\mathbb{L}_{\mathrm{m}(k)}$ which has k injections, but no booleans, numbers, or pairs.

DEFINITION 5.1 $\mathbb{L}_{\mathrm{m}(k)}$ has structure $\langle \mathbb{E}_{\mathrm{m}(k)}, \mathbb{V}_{\mathrm{m}(k)}, \mathbb{O}_{\mathrm{m}(k)}, \mapsto_{\mathrm{m}(k)} \rangle$, where

$$\mathbb{O} = \mathbb{O}_{\mathrm{inj}(k)} \cup \{\mathtt{ref}(e), \mathtt{set}(e, e'), \mathtt{get}(e), \mathtt{iscell}(e)\}$$

$\mathtt{ref}(e)$ creates a new memory cell with e as the initial value, $\mathtt{set}(e, e')$ sets a cell e with new value e', and $\mathtt{get}(e)$ gets the value from cell e. We let $a;\ b$ abbreviate sequencing, $(\lambda x.b)(a)$ for x fresh.

We hereafter take k to be an arbitrary fixed value and refer to this language as $\mathbb{L}_{\mathrm{m}}$. The previous section gives evidence that a family of injections captures the complexity of other simple features such as booleans, numbers, pairs, and objects. In our embedding of objects into $\mathbb{L}_{\mathrm{bpn}(k)}$ we preserved these features because the embedding was homomorphic. In the memory case, no homomorphic embedding can be defined.

LEMMA 5.2 There is no embedding $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{m}(k)} \to \mathbb{L}_{\mathrm{bpn}(k+j+1)}$ for any j, k that is homomorphic in $\mathbb{O}_{\mathrm{inj}(k)}$.

PROOF: The basic idea of the proof is the observation that a homomorphic embedding would force functions in the memory language to have no side-effects. $(\lambda x.y(0))(y(0)) \cong_{\mathrm{bpn}(k+j+1)} y(0)$, so if a homomorphic map existed, by lifting we would have $(\lambda x.y(0))(y(0)) \cong_{\mathrm{m}} y(0)$ but $y = \lambda x.\mathtt{set}(z, \mathtt{inj}_i(\mathtt{get}(z)))$ causes this to fail. □

$\mathbb{L}_{\mathrm{m}}$ is embedded into $\mathbb{L}_{\mathrm{bpn}(k+1)}$ by the following parametric embedding, which passes the memory to all computations and computes (value, memory)-pairs as

result. $\mathtt{inj}_{k+1}$ is used to label cell numbers. The memory is a function from cell numbers to values; memory location 0 holds the next free cell address.

DEFINITION 5.3 Parametric embedding $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{m}} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ is defined as $C_{\mathrm{init}} = \bullet\,(\,\lambda x.0\,)$ and an embedding of expressions as follows.

$$
\begin{array}{rcl}
[\![x]\!] & = & \lambda m.\mathtt{pr}(x, m) = C_{\mathrm{var}}[x] \\
[\![\mathtt{ref}(e)]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(v, m_1) = [\![e]\!](m)\ \mathtt{in} \\
 & & \quad \mathtt{pr}(\mathtt{inj}_{k+1}(m_1(0)), \lambda c'.\mathtt{if}(\mathtt{iszero}(c'), \mathtt{succ}(m_1(0)), \\
 & & \quad\quad \mathtt{if}(\mathtt{nateq}(c', m_1(0)), v, m_1(c')))) \\
[\![\mathtt{set}(e, e')]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(c, m_1) = [\![e]\!](m)\ \mathtt{in} \\
 & & \quad \mathtt{let}\ \mathtt{pr}(v, m_2) = [\![e']\!](m_1)\ \mathtt{in} \\
 & & \quad \mathtt{pr}(v, \lambda c'.\mathtt{if}(\mathtt{nateq}(c', \mathtt{out}_{k+1}(c)), v, m_2(c'))) \\
[\![\mathtt{get}(e)]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(c, m_1) = [\![e]\!](m)\ \mathtt{in}\ \mathtt{pr}(m_1(\mathtt{out}_{k+1}(c)), m_1) \\
[\![\mathtt{iscell}(e)]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(v, m_1) = [\![e]\!](m)\ \mathtt{in}\ \mathtt{pr}(\mathtt{is}_{k+1}(v), m_1) \\
[\![\lambda x.e]\!] & = & \lambda m.\mathtt{pr}(\lambda x.[\![e]\!], m) \\
[\![\mathtt{op}(e)]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(v, m_1) = [\![e]\!](m)\ \mathtt{in}\ \mathtt{pr}(\mathtt{op}(v), m_1) \\
 & & \text{for all other unary operators } \mathtt{op} \in \mathbb{O}_{\mathrm{m}} \\
[\![e\,(e')]\!] & = & \lambda m.\mathtt{let}\ \mathtt{pr}(f, m_1) = [\![e]\!](m)\ \mathtt{in} \\
 & & \quad \mathtt{let}\ \mathtt{pr}(v, m_2) = [\![e']\!](m_1)\ \mathtt{in}\ f\,(v)\,(m_2)
\end{array}
$$

This encoding is clearly not fully abstract: $\mathbb{L}_{\mathrm{bpn}(k+1)}$ contexts will be able to inspect and alter the contents of arbitrary memory cells.

LEMMA 5.4 $[\![\cdot]\!] \in \mathbb{L}_{\mathrm{m}} \to \mathbb{L}_{\mathrm{bpn}(k+1)}$ is not fully abstract.

PROOF: $\lambda x.e\,(\,e\,) \cong_{\mathrm{bpn}(k+1)} e$ for x not free in e, but if e has a cumulative effect (such as $e = \mathtt{set}(y, \mathtt{inj}_i(\mathtt{get}(y)))$) the equation will fail to be $\cong_{\mathrm{m}}$. □

This parametric embedding is only sometimes useful for establishing equivalences $\cong_{\mathrm{m}}$: since it is not homomorphic, there is no lifting of equations possible. The one tool it provides is the ability to prove $a \cong_{\mathrm{m}} b$ by showing $[\![a]\!] \cong_{\mathrm{bpn}(k+1)} [\![b]\!]$. This technique is useful for principles that do not require memory locality, the source of lack of full abstraction in the translation.

5.1 Direct Memory Semantics

Since working in memory-passing style is difficult, hereafter we will work over a direct evaluation semantics for $\mathbb{L}_{\mathrm{m}}$, using notation close to (Honsell, Mason, Smith, and Talcott 1995; Mason and Talcott 1991). Most of the proofs in this section are direct generalizations of the functional proofs from (Mason, Smith, and Talcott 1996) to a language with memories. Very little extra reasoning is needed to handle the memory. The projection functions, defined in the following section, do require considerable extra work in the memory case.

A memory context Γ is defined as a context

$$\mathtt{let}\ c_1 = \mathtt{ref}(\mathtt{etrue}) \ldots \mathtt{let}\ c_n = \mathtt{ref}(\mathtt{etrue})\ \mathtt{in}\ \mathtt{set}(c_1, v_1) \ldots ; \mathtt{set}(c_n, v_n);\ \bullet$$

for distinct $c_1, \ldots, c_n$. Γ may equivalently be viewed as a finite map from variables to values; here $\mathrm{Dom}(\Gamma) = c_1, \ldots, c_n$. To distinguish variables used as cell names from other variables, we define $\mathbb{X}_{\text{cell}} \subseteq \mathbb{X}$ such that $\mathbb{X}_{\text{cell}}$ and $\mathbb{X}_\lambda = \mathbb{X} - \mathbb{X}_{\text{cell}}$ are both countably infinite. Invariably, $\mathrm{Dom}(\Gamma) \subseteq \mathbb{X}_{\text{cell}}$, and λ-bound variables are in $\mathbb{X}_\lambda$. α-conversion of cell variables is not allowed. We use shorthand $\Gamma\{c = v\}$ to indicate an extension to Γ mapping cell c to value v.

For brevity we will define some notions as extensions of the notions defined in the presentation of $\mathbb{L}_{\text{inj}(k)}$ of Section 3, reading those definitions with $\mathbb{E}_{\text{m}}$ in place of $\mathbb{E}_{\text{inj}(k)}$. The values $\mathbb{V}_{\text{m}}$ are the same cases as for $\mathbb{L}_{\text{inj}(k)}$, and the reduction contexts $\mathbb{R}_{\text{m}}$ include the same cases plus $\mathtt{ref}(\bullet) \cup \mathtt{set}(\bullet, \mathbb{E}_{\text{m}}) \cup \mathtt{set}(\mathbb{V}_{\text{m}}, \bullet) \cup \mathtt{get}(\bullet) \cup \mathtt{iscell}(\bullet)$. Single-step evaluation is a map $\Gamma[R[e]] \mapsto^1_{\text{m}} \Gamma'[R[e']]$ for $R \in \mathbb{R}_{\text{m}}$, e a redex and e' its contractum.

DEFINITION 5.5 ($\mapsto^1_{\text{m}}$, $\mapsto_{\text{m}}$) $\mapsto_{\text{m}}$ is the transitive, reflexive closure $\mapsto^1$:

(fun)	$\Gamma[a]$	$\mapsto^1$	$\Gamma[b]$
	where $a \mapsto^1_{\text{inj}(k)} b$ is a case of Definition 3.4		
(ref)	$\Gamma[R[\mathtt{ref}(v)]]$	$\mapsto^1$	$\Gamma\{c = v\}[R[c]]$ for fresh $c \in \mathbb{X}_{\text{cell}}$
(set)	$\Gamma\{c = v_0\}[R[\mathtt{set}(c, v)]]$	$\mapsto^1$	$\Gamma\{c = v\}[R[v]]$
(get)	$\Gamma\{c = v\}[R[\mathtt{get}(c)]]$	$\mapsto^1$	$\Gamma\{c = v\}[R[v]]$
(iscell-t)	$\Gamma[R[\mathtt{iscell}(c)]]$	$\mapsto^1$	$\Gamma[R[\mathtt{etrue}]]$ where $c \in \mathbb{X}_{\text{cell}}$
(iscell-f)	$\Gamma[R[\mathtt{iscell}(v)]]$	$\mapsto^1$	$\Gamma[R[\mathtt{efalse}]]$ where $v \notin \mathbb{X}$

The **ciu** ordering is defined for $\mathbb{L}_{\text{m}}$ as follows.

DEFINITION 5.6 (CIU ORDERING, $\precsim^{\text{ciu}}_{\text{m}}$) For all a and b, $a \precsim^{\text{ciu}}_{\text{m}} b$ iff for all Γ, σ, R such that $\Gamma[R[a\,[\sigma]]]$ and $\Gamma[R[b\,[\sigma]]]$ are closed, if $\Gamma[R[a\,[\sigma]]] \downarrow$ then $\Gamma[R[b\,[\sigma]]] \downarrow$.

THEOREM 5.7 (CIU) $a \precsim_{\text{m}} b$ iff $a \precsim^{\text{ciu}}_{\text{m}} b$.

For a proof, see (Honsell, Mason, Smith, and Talcott 1995). The theorem also follows from set ordering **ciu**, proved below as Theorem 5.10, by Lemma 2.6 (iii). Applicative bisimulation equivalences may be defined for languages with memories (Ritter and Pitts 1995) and may be effectively used to directly establish equivalences. They (to lesser degree) suffer the same problem of lack of locality as does $[\![a]\!] \cong_{\text{bpn}(k+1)} [\![b]\!]$, and so are not fully abstract.

An important property that holds for $\precsim_{\text{m}}$ is extensionality. It seems not to have been stated or proved before in the literature so we give a proof.

THEOREM 5.8 ($\precsim_{\text{m}}$ EXTENSIONALITY) For all a_0 and a_1, $\lambda x.a_0 \precsim_{\text{m}} \lambda x.a_1$ if and only if $(\lambda x.a_0)(v) \precsim_{\text{m}} (\lambda x.a_1)(v)$ for all values v.

PROOF: $\Rightarrow$ is direct from the pre-congruence of $\precsim_{\text{m}}$. For $\Leftarrow$, assume for all v that $\lambda x.a_0(v) \precsim_{\text{m}} \lambda x.a_1(v)$ and show $\lambda x.a_0 \precsim_{\text{m}} \lambda x.a_1$. It suffices to fix v to be x; then by β-value, $a_0 \precsim_{\text{m}} a_1$ and so by congruence, $\lambda x.a_0 \precsim_{\text{m}} \lambda x.a_1$. □

It is perhaps surprising that extensionality holds, because extensionality shows that it suffices to test a function without iterative application. If the function is a closure with local state it will not have the same value when executed a second time. However, since the theorem is proved for λ-values only, these cannot have any local state. Extensionality fails for imperative objects (closures), because they may allocate their own local store. Consider the simple object defined as follows.

$$o = \texttt{let } c = \texttt{ref}(\texttt{etrue}) \texttt{ in } \lambda z.\texttt{set}(c, \texttt{inj}_1(\texttt{get}(c)))$$

$o(\,v\,) \cong_{\mathrm{m}} \texttt{inj}_1(\texttt{etrue})$ for any v by computing, so if o was an extensional object, $o \cong_{\mathrm{m}} \lambda z.\texttt{inj}_1(\texttt{etrue})$ should hold but it clearly does not.

As was the case for $\mathbb{L}_{\mathrm{inj}(k)}$, the $\precsim_{\mathrm{m}}$-directed set ordering $\{\cdot\} \precsim_{\mathrm{m}} \{\cdot\}$ also has a **ciu** characterization which is critical for proving facts about $\{\cdot\} \precsim_{\mathrm{m}} \{\cdot\}$.

DEFINITION 5.9 (CIU SET ORDERING, $\precsim_{\mathrm{m}}^{\mathrm{ciu}}$) For $\precsim_{\mathrm{m}}$-directed sets of expressions A and B, $A \precsim_{\mathrm{m}}^{\mathrm{ciu}} B$ if and only if for all $a \in A$ and for all Γ, σ, R such that $\Gamma[R[A\,[\sigma]]]$ and $\Gamma[R[B\,[\sigma]]]$ are sets of closed expressions, if $\Gamma[R[a\,[\sigma]]]\downarrow$ then there exists a $b \in B$ such that $\Gamma[R[b\,[\sigma]]]\downarrow$.

The **ciu** theorem is

THEOREM 5.10 ($\precsim_{\mathrm{m}}^{\mathrm{ciu}}$ CIU) $A \precsim_{\mathrm{m}} B$ iff $A \precsim_{\mathrm{m}}^{\mathrm{ciu}} B$.

This proof does not appear in previous papers and so is given here. The proof here is a direct combination of the proof of $\{\cdot\} \precsim \{\cdot\}$ **ciu** for a functional call-by-value language in (Mason, Smith, and Talcott 1996) and **ciu** for a memory-based language in (Honsell, Mason, Smith, and Talcott 1995). The proof may be factored into three Lemmas. The ($\Rightarrow$) direction is not difficult, since $\precsim_{\mathrm{m}}^{\mathrm{ciu}}$ has a smaller collection of contexts to distinguish expressions than $\{\cdot\} \precsim_{\mathrm{m}} \{\cdot\}$ has. ($\Leftarrow$) is the difficult direction. This proof uses the observation that it suffices to show $\precsim_{\mathrm{m}}^{\mathrm{ciu}}$ is a pre-congruence. To establish this, we prove lemmas that establish pre-congruence for single constructors: non-λ operators $\mathbb{O}_{\mathrm{m}}$ (Lemma 5.11) and λx (Lemma 5.12) may be placed around sets of expressions while preserving $\precsim_{\mathrm{m}}^{\mathrm{ciu}}$.

LEMMA 5.11 ($\precsim_{\mathrm{m}}^{\mathrm{ciu}}$ OPERATOR CIU) If $A \precsim_{\mathrm{m}}^{\mathrm{ciu}} B$ holds, then

$$\texttt{op}(\bar{c}, A, \bar{d}) \precsim_{\mathrm{m}}^{\mathrm{ciu}} \texttt{op}(\bar{c}, B, \bar{d})$$

for any $\texttt{op} \in \mathbb{O}_{\mathrm{m}}$ that is not `lambda`.

PROOF: We consider the case of an arbitrary binary operator `op`; the simpler unary case should be uniformly apparent from this case. It suffices to show the two cases

$\Gamma[R[\texttt{op}(a\,[\sigma], c))]]\downarrow$ implies there exists a $b \in B$ such that $\Gamma[R[\texttt{op}(b\,[\sigma], c)]]\downarrow$

and

$$\Gamma[R[\mathtt{op}(c, a\,[\sigma]))]]\downarrow \text{ implies there exists a } b \in B \text{ such that } \Gamma[R[\mathtt{op}(c, b\,[\sigma])]]\downarrow$$

We proceed by induction on the length of the computation of the assumption. Assume the statements are true for all shorter computations. In the first case above define $R_0 = R[\mathtt{op}(\bullet, c)]$, and in the second case when c is a value, define $R_0 = R[\mathtt{op}(c, \bullet)]$, and the conclusion follows directly by assumption. So, we may concentrate on the second case when c is not a value. In this case we have a reduction context

$$R_0 = R[\mathtt{op}(\bullet, a\,[\sigma])],$$

so by computing

$$\Gamma[R[\mathtt{op}(c, a\,[\sigma])]] \mapsto^1 \Gamma'[R[\mathtt{op}(c', a\,[\sigma])]],$$

which is an instance of the induction hypothesis since this computation will terminate in one fewer steps. □

LEMMA 5.12 ($\sqsubseteq^{\mathrm{ciu}}_{\sim\mathrm{m}}$ LAMBDA CIU) If $A \sqsubseteq^{\mathrm{ciu}}_{\sim\mathrm{m}} B$, then $\lambda x.A \sqsubseteq^{\mathrm{ciu}}_{\sim\mathrm{m}} \lambda x.B$.

PROOF: It suffices to assume expressions in A and B contain at most x free, the conclusion then follows by definition of $\sqsubseteq^{\mathrm{ciu}}_{\sim\mathrm{m}}$. For arbitrary R and Γ, show for fixed $a \in A$

$$\Gamma[R[\lambda x.a]]\downarrow \text{ implies there exists a } b \in B \text{ such that } \Gamma[R[\lambda x.b]]\downarrow.$$

We may generalize this statement to

$$\Gamma[e]\,[\lambda x.a/z]\downarrow \text{ implies there exists a } b \in B \text{ such that } \Gamma[e]\,[\lambda x.b/z]\downarrow,$$

for free variables of e coming only from $\mathrm{Dom}(\Gamma) \cup \{z\}$. The original goal follows by letting $e = R[z]$. Proceed by induction on the length of the computation of the assumption. Consider whether $\Gamma[e]$ is uniform in z, i.e. whether

$$\Gamma[e] \mapsto^1 \Gamma'[e']$$

for some $\Gamma'[e']$. If it is uniform, then

$$\Gamma[e]\,[\lambda x.e''/z] \mapsto^1 \Gamma'[e']\,[\lambda x.e''/z], \text{ for all } e'',$$

and the result follows by the induction hypothesis.

Consider the case where $\Gamma[e]$ is stuck, i.e. does not reduce. Since $\Gamma[e]\,[\lambda x.a/z]\downarrow$, it does not get stuck when a λ-value is substituted for z. By the evaluation rules, replacing z with a λ-value causes a stuck computation to become un-stuck in two cases. The first is if the redex is $\mathtt{is}_i(z)$ or $\mathtt{iscell}(z)$; but these cases are still uniform for any λ-value and reasoning analogous to the previous uniform case applies. The only other non-uniform case is where $e = R[z(v)]$ for some R, v

which have free variables from $\mathrm{Dom}(\Gamma) \cup \{z\}$. By the form of the β-value reduction rule, we then have the following:

$$\Gamma[R[(\lambda x.e')(v)]] \mapsto^1 \Gamma[R[e'[v/x]]],$$

for all expressions e', v, R. In particular, it holds for e' being a or any $b \in B$. It thus suffices to show

$$\text{there exists a } b \in B \text{ such that } \Gamma[R[b[v/x]]][\lambda x.b/z]\downarrow .$$

By the induction hypothesis,

$$\text{there exists a } b' \in B \text{ such that } \Gamma[(R[a[v/x]])][\lambda x.b'/z]\downarrow .$$

Then by the assumption $A \precsim^{\mathrm{ciu}}_{\mathrm{m}} B$, a above may be replaced by some $b'' \in B$ (take the substitution in the definition of $\precsim^{\mathrm{ciu}}_{\mathrm{m}}$ to be $[v[\lambda x.b'/z]/x]$), giving

$$\text{there exists a } b'' \in B \text{ such that } \Gamma[(R[b''[v/x]])][\lambda x.b'/z]\downarrow .$$

By the directedness of B, we can find a b such that $b', b'' \precsim_{\mathrm{m}} b$, and this means first that

$$\Gamma[(R[b''[v/x]])][\lambda x.b/z]\downarrow .$$

Now by the simple fact that $\precsim_{\mathrm{m}}$ respects value substitution, $b''[v'/x] \precsim_{\mathrm{m}} b[v'/x]$ for $v' = v[\lambda x.b/z]$, so

$$\Gamma[(R[b[v/x]])][\lambda x.b/z]\downarrow .$$

□

Iteratively applying the two previous lemmas then allows an arbitrary context to be constructed around sets A and B one operator at a time:

LEMMA 5.13 ($\precsim^{\mathrm{ciu}}_{\mathrm{m}}$ PRE-CONGRUENCE) $\precsim^{\mathrm{ciu}}_{\mathrm{m}}$ is a pre-congruence, $A \precsim^{\mathrm{ciu}}_{\mathrm{m}} B$ implies $C[A] \precsim^{\mathrm{ciu}}_{\mathrm{m}} C[B]$.

And, from $\precsim^{\mathrm{ciu}}_{\mathrm{m}}$ pre-congruence, Theorem 5.10 directly follows.

LEMMA 5.14 (FIXED POINT) For a functional f, $\{\mathtt{fix}(f)\} \cong_{\mathrm{m}} \{f^n \mid n \in \mathbb{N}\}$.

PROOF: Without loss of generality assume the free variables of f are cell variables only, for from this case the result follows for arbitrary f by Theorem 5.10. The $\precsim_{\mathrm{m}}$ direction follows by induction on n; consider then proving $\precsim_{\mathrm{m}}$. First note $\mathtt{fix}(f) \cong_{\mathrm{m}} u(u)$ where $u = \lambda x.\lambda z.f(x(x))(z)$, so it suffices to show $\{u(u)\} \precsim_{\mathrm{m}} \{f^k \mid k \in \mathbb{N}\}$. Expanding definitions, the desired result is to show for all a with free variables from $\{x\} \cup \mathrm{Dom}(\Gamma)$, $\Gamma[a][u(u)/x]\downarrow$ implies $\Gamma[a][f^k/x]\downarrow$ for some k. Assume $\Gamma[a][u(u)/x]\downarrow$, proceed by induction on the length of this computation to show the above statement. Consider the next step of computation performed on $\Gamma[a][u(u)/x]$. If the step is uniform in $u(u)$, the conclusion follows

directly by induction hypothesis. Then, consider non-uniform steps; all such cases can easily be seen to be of the form

$$\Gamma[a]\,{}_{[u(u)/x]} = \Gamma[R[u\,(\,u\,)]]\,{}_{[u(u)/x]} \mapsto_1 \Gamma[R[\lambda z.f(u\,(\,u\,))(z)]]\,{}_{[u(u)/x]},$$

we show $\Gamma[R[f^k]]\,{}_{[f^k/x]}\!\downarrow$ for some k. By the induction hypothesis,

$$\Gamma[R[\lambda z.f(f^{k_0})(z)]]\,{}_{[f^{k_0}/x]}\!\downarrow$$

for some k_0, so since $f^{k_0} \sqsubseteq_{\mathrm{m}} f^{k_0+1}$ and $\lambda z.f(f^{k_0})(z) \cong_{\mathrm{m}} f^{k_0+1}$ by extensionality, $\Gamma[R[f^{k_0+1}]]\,{}_{[f^{k_0+1}/x]}\!\downarrow$, and letting k be k_0+1, the desired conclusion has been reached. □

5.2 Memory Projections

We will hereafter informally use numbers, pairs, and lists as $\mathbb{L}_{\mathrm{m}}$ syntax, taking numbers and pairs to be encoded as in the embedding of $\mathbb{L}_{\mathrm{bpn}}$ of Definition 4.3, and (functional) lists and list operations, `nil`/`cons`/`carcdr`/`isnil`/`mapcar`, encoded via the standard pair-based encoding. We will define `member` for lists of cells below. Syntactic projections π_{m}^n may be defined as follows.

DEFINITION 5.15 (PROJECTIONS π_{m}^n)

$$\begin{array}{l}
\pi_{\mathrm{m}} = \lambda y.\lambda z.\lambda x. \\
\qquad \mathtt{eif}(\mathtt{is}_1(x), \mathtt{inj}_1(y\,(z)\,(\mathtt{out}_1(x)\,)), \\
\qquad \mathtt{eif}(\mathtt{is}_2(x), \mathtt{inj}_2(y\,(z)\,(\mathtt{out}_2(x)\,)), \ldots, \\
\qquad \mathtt{eif}(\mathtt{is}_k(x), \mathtt{inj}_k(y\,(z)\,(\mathtt{out}_k(x)\,)), \\
\qquad \mathtt{eif}(\mathtt{iscell}(x), \mathtt{set}(w, \mathtt{cons}(x, \mathtt{get}(w))); \\
\qquad\quad \mathtt{eif}(\mathtt{member}(x,z), x, \mathtt{set}(x, y\,(\mathtt{cons}(x,z))\,(\mathtt{get}(x)\,)))), \\
\qquad \lambda x_0.\mathtt{mapcar}(\lambda x.y\,(\mathtt{nil})\,(x\,), \mathtt{get}(w)); \\
\qquad\quad \mathtt{let}\ r = y\,(\mathtt{nil})\,(x\,(y\,(\mathtt{nil})\,(x_0\,))\,)\ \mathtt{in} \\
\qquad\quad \mathtt{mapcar}(\lambda x.y\,(\mathtt{nil})\,(x\,), \mathtt{get}(w));\ r)\ldots)) \\
\\
\mathtt{member}(x,l) = \mathtt{fix}(\lambda f.\lambda l.\mathtt{eif}(\mathtt{isnil}(l), \mathtt{efalse}, \\
\qquad \mathtt{eif}(\mathtt{celleq}(x, \mathtt{car}(l)), \mathtt{etrue}, f\,(\mathtt{cdr}(l)\,))\,(l\,) \\
\mathtt{celleq}(x,y) = \mathtt{let}\ x_0 = \mathtt{get}(x)\ \mathtt{in}\ \mathtt{let}\ y_0 = \mathtt{get}(y)\ \mathtt{in} \\
\qquad \mathtt{set}(x, \mathtt{efalse});\ \mathtt{set}(y, \mathtt{etrue}); \\
\qquad \mathtt{let}\ r = \mathtt{get}(x)\ \mathtt{in}\ \mathtt{set}(x, x_0);\ \mathtt{set}(y, y_0);\ r \\
\pi_{\mathrm{m}}^0 = \lambda x.\mathtt{bot} \\
\pi_{\mathrm{m}}^{n+1} = \mathtt{let}\ w = \mathtt{ref}(\mathtt{nil})\ \mathtt{in}\ \pi_{\mathrm{m}}\,(\pi_{\mathrm{m}}^n\,)\,(\mathtt{nil}\,) \\
\pi_{\mathrm{m}}^{\infty} = \mathtt{let}\ w = \mathtt{ref}(\mathtt{nil})\ \mathtt{in}\ \mathtt{fix}\,(\pi_{\mathrm{m}}\,)\,(\mathtt{nil}\,)
\end{array}$$

The difficult question is what the projection operation should do with a memory cell. The above projections will project the contents of any memory cell encountered. If there is cyclic data in the memory, such as a cell x containing $\mathtt{inj}_2(x)$, we must not

repeatedly project x, for this process will loop forever. The extra z parameter here, not found in the functional projection functions, serves the purpose of accumulating cells already encountered, and preventing such cells from being projected again. Note that projection of cell x containing $\lambda y.x$ causes no looping problem because the projection operation will halt at the λ. For this reason, at this point z is reset.

The global cell list in reference w serves to close a "back-door" communication channel. Note that w is free in π_{m} and becomes bound in π_{m}^{n+1} and $\pi_{\mathrm{m}}^{\infty}$ definitions. The cells in this list are cells that "entered" or "exited" this projection at some point in the computation history. Subexpression

$$\mathtt{mapcar}(\lambda x.y\,\mathtt{(}\,x\,\mathtt{)(}\,z\,\mathtt{)},\mathtt{get}(w))$$

above serves to project all the cells accumulated thus far in w. Without w and these additional projections, cells would only be projected when they are explicitly passed to or returned from a function, and thus a cell passed to a projected function could on successive calls to the function serve as a "back-door" communication channel if the function remembers this cell name locally. We give an example to clarify this point. A wrapper around a function f of the form

$$\begin{array}{rcl} z & = & \mathtt{let}\ c = \mathtt{ref}(\mathtt{inj}_1(\lambda x.x))\ \mathtt{in} \\ & & \lambda y.\mathtt{eif}(\mathtt{is}_1(\mathtt{get}(c)),\mathtt{set}(c,y),\mathtt{set}(\mathtt{get}(c),f\,\mathtt{(}\,\mathtt{get}(\mathtt{get}(c))\mathtt{)})) \end{array}$$

would then allow f to be computed even when z is projected: first a cell could be passed in to z which serves as a communication channel to the context that would not be subsequently projected. Consider

$$\begin{array}{rcl} f' & = & \lambda x.\mathtt{let}\ z = \pi_{\mathrm{m}}^3\,\mathtt{(}\,z\,\mathtt{)}\ \mathtt{in} \\ & & \mathtt{let}\ y = \mathtt{ref}(\lambda x.\mathtt{bot})\ \mathtt{in}\ z\,\mathtt{(}\,y\,\mathtt{)};\ \mathtt{set}(y,x);\ z\,\mathtt{(}\,\lambda x.x\,\mathtt{)};\ \mathtt{get}(y) \end{array}$$

—$f \cong_{\mathrm{m}} f'$ would hold if nonlocal cell projection were not a component of the definition of π_{m} above.

We may prove π_{m}^n possesses the finite approximation property:

THEOREM 5.16 Finite Approximation, Theorem 3.14, is provable for the projections π_{m}^n: $\{\pi_{\mathrm{m}}^n\,\mathtt{(}\,a\,\mathtt{)} \mid n \in \mathbb{N}\} \cong_{\mathrm{m}} \{a\}$.

The proof of this Theorem parallels the proof for $\mathbb{L}_{\mathrm{inj}(k)}$ of Section 3.2. More details of proofs are provided here since the memory changes some aspects of the proof in a nontrivial way. Hereafter the m subscript on projections π is implicit.

LEMMA 5.17 (ELEMENTARY $\pi_{\mathrm{m}}^n/\pi_{\mathrm{m}}^{\infty}$ PROPERTIES) The elementary π^n/π^{∞} properties of Lemma 3.13 all hold when lifted to $\mathbb{L}_{\mathrm{m}}$.

Following $\mathbb{L}_{\mathrm{inj}(k)}$, we define $\tau(a)$ and $\tau(R)$ to characterize how the projections percolate into expressions. The only addition is the projections may percolate into the memory Γ and so $\tau(\Gamma)$ also needs to be defined. The definitions of $\tau(a)$ and

$\tau(R)$ may be directly lifted from $\mathbb{L}_{\mathrm{inj}(k)}$ (cells are in fact variables, so no extra case is needed there). $\tau(\Gamma)$ is defined as replacing each cell value v in Γ with $\tau(v)$. Observe that $\tau(\Gamma)$ is always a legal memory context since it stores only values. Basic properties of $\tau(e)$ sets include the following.

LEMMA 5.18 (i) For a with all free variables bound by memory context Γ, $\tau(\Gamma)[\tau(R)[\pi^\infty(\tau(a))]]\downarrow \Leftrightarrow \tau(\Gamma)[\tau(R)[\tau(a)]]\downarrow$.

(ii) $\tau(a) \sqsubseteq_{\mathrm{m}} a$, $\tau(R[x]) \sqsubseteq_{\mathrm{m}} R[x]$, and $\tau(\Gamma[a])\downarrow \Rightarrow \Gamma[a]\downarrow$.

(iii) $\tau(R[b]) = \tau(R)[\tau(b)]$, and $\tau(a\,[v/x]) = \tau(a)\,[\tau(v)/x]$.

PROOF: (i), $\Rightarrow$ follows from Lemma 5.17, (prune). For $\Leftarrow$, first observe it suffices to consider the case of a being a value by computing. The structure of values in this language can be viewed as a chain of chains terminating in a λ, described as follows: outermost, the value is $\mathtt{inj}_{i_1}(\ldots \mathtt{inj}_{i_n}(v)\ldots)$ where v is a λ or a cell; if it is a cell, the contents of the cell, $\tau(\Gamma)(v)$, in turn must have a similar chain structure. This chain of chains must eventually terminate in a λ, or in a cell already encountered previously on the chain. An induction on the structure of this chain establishes that the π^∞ operation here has no effect.

(ii) follows from Lemma 5.17 (prune), and (iii) is direct from the definition of τ. □

LEMMA 5.19 (IDENTITY OF π^∞_{m}) $\pi^\infty \cong_{\mathrm{m}} \lambda x.x$

PROOF: The $\sqsubseteq_{\mathrm{m}}$ direction follows from Lemma 5.17 (prune) and extensionality. For the $\sqsupseteq_{\mathrm{m}}$ direction, we successively rephrase the statement five times. It suffices to show for all a with only cell variables free that $\Gamma[R[a]]\downarrow \Rightarrow \Gamma[R[\pi^\infty(a)]]\downarrow$ by the **ciu** and extensionality theorems. For this it then suffices to show $\Gamma[R[a]]\downarrow \Rightarrow \tau(\Gamma)[\tau(R)[\pi^\infty(\tau(a))]]\downarrow$ by Lemma 5.18 (ii). And, by Lemma 5.18 (i) it then suffices to show $\Gamma[R[a]]\downarrow \Rightarrow \tau(\Gamma)[\tau(R)[\tau(a)]]\downarrow$ So, it suffices to show $\Gamma[a_0]\downarrow \Rightarrow \tau(\Gamma)[\tau(a_0)]\downarrow$ by Lemma 5.18 (iii). And lastly, to show this it suffices to show $\Gamma_0[a_0] \mapsto^1 \Gamma_1[a_1] \Rightarrow \tau(\Gamma_0)[\tau(a_0)]\downarrow \Leftrightarrow \tau(\Gamma_1)[\tau(a_1)]\downarrow$, for the conclusion then follows by induction on computation length and the observation that $\tau(v)$ is a value for any value v.

So, assume $\Gamma_0[a_0] \mapsto^1 \Gamma_1[a_1]$, show $\tau(\Gamma_0)[\tau(a_0)]\downarrow \Leftrightarrow \tau(\Gamma_1)[\tau(a_1)]\downarrow$. Consider this step of computation; $a_0 = R[a]$ for some redex a; proceed by cases on the form of a.

If $a = \mathtt{app}(\lambda x.c, v)$, then $a_1 = R[c\,[v/x]]$. By inspection of the definitions of $\tau(a)$ and $\tau(R)$, $\tau(\Gamma_0)[\tau(a_0)]$ must be of the form

$$\tau(\Gamma_0)[\tau(R)[\pi^\infty(\mathtt{app}(\pi^\infty \circ\ \lambda x.\tau(c)\ \circ \pi^\infty, \tau(v)))]].$$

Computing from this point yields

$$\begin{array}{ll} & \tau(\Gamma_0)[\tau(R)[\pi^\infty(\mathtt{app}(\pi^\infty \circ\ \lambda x.\tau(c)\ \circ \pi^\infty, \tau(v)))]]\downarrow \\ \Leftrightarrow & \tau(\Gamma_0)[\tau(R)[\pi^\infty(\pi^\infty(\mathtt{app}(\lambda x.\tau(c), \pi^\infty(\tau(v)))))]]\downarrow \\ \Leftrightarrow & \tau(\Gamma_0)[\tau(R)[\pi^\infty(\mathtt{app}(\lambda x.\tau(c), \tau(v)))]]\downarrow \text{ by Lemmas 5.17 (idemp), 5.18 (i)} \\ \Leftrightarrow & \tau(\Gamma_0)[\tau(R)[\pi^\infty(\tau(c)[\tau(v)/x])]]\downarrow \\ \Leftrightarrow & \tau(\Gamma_0)[\tau(R)[\tau(c)[\tau(v)/x]]]\downarrow \text{ by Lemma 5.18 (i) and (iii)} \\ \Leftrightarrow & \tau(\Gamma_0)[\tau(R[c[v/x]])]\downarrow \text{ by Lemma 5.18 (iii).} \end{array}$$

If a is any other redex except a memory operation, the proof is similar to the previous case. For the memory operations, the proof is somewhat similar but requires a bit of extra reasoning; consider redex $\mathtt{set}(x, v)$. The memory cell x in Γ_1 will then have value v, whereas $\mathtt{set}(x, \tau(v))$ will place $\tau(v)$ as x's value, precisely what $\tau(\Gamma_1)$ should be by its definition. □

Unfortunately these projections do not produce finite elements. The projection operations π can force the domain and range of a function to be statically finite, but there are still infinitely many different histories this function can have on successive invocations.

LEMMA 5.20 (π_{m} FINITENESS FAILURE) $\{a \mid a \text{ is closed and } a \cong_{\mathrm{m}} \pi^3(a)\}$ contains infinitely many $\cong_{\mathrm{m}}$-distinct expressions.

PROOF: For each n,

$$\mathtt{let}\ x = \mathtt{ref}(n)\ \mathtt{in}\ \lambda x.\mathtt{if}(\mathtt{iszero}(\mathtt{set}(x, \mathtt{pred}(\mathtt{get}(x)))), \mathtt{bot}, 0)$$

is a distinct constant function which only may be used up to n times before diverging. □

This suggests that some aspect of history must be included in a finite characterization of memory-based languages. There is one additional incompleteness. These projected expressions are also not finite in the operational sense, namely a computation of $\pi^n(a)$ could compute forever without attempting to compute $\pi^0(v)$ for some v. In particular, a memory-based fixed point (defined as a function in a cell c which in its body retrieves the function in c, *i.e.* itself, and invokes it) could compute forever even if every subexpression is projected. Consider for example the evaluation of

$$\Gamma\{c = \pi^n_{\mathrm{m}} \circ \lambda y.\pi^n_{\mathrm{m}}(\mathtt{get}(\pi^{n+1}_{\mathrm{m}}(c))(0))\}[\pi^n_{\mathrm{m}}(\mathtt{get}(\pi^{n+1}_{\mathrm{m}}(c))(0))].$$

This computation will compute infinitely along the sequence of states

$$\Gamma\{c = \pi^n_{\mathrm{m}} \circ \lambda y.\pi^n_{\mathrm{m}}(\mathtt{get}(\pi^{n+1}_{\mathrm{m}}(c))(0))\}[\pi^n_{\mathrm{m}}(\ldots\pi^n_{\mathrm{m}}(\mathtt{get}(\pi^{n+1}_{\mathrm{m}}(c))(0))\ldots)].$$

Any functional fixed point which uses the projected fixed point combinator $\pi^n(\mathtt{fix})$ will not suffer from this problem: the bound n will be the maximum number of recursive calls of $\pi^n(\mathtt{fix})(f)$ for any functional f.

5.3 Toward Finite Memory Projections

We now outline a potential solution to the above problems. The number of times a projected function can be successively invoked is limited to a fixed number, removing the infinitude uncovered in Lemma 5.20. This is implemented by modifying projections at level n to use a unique counter cell for each function that counts the number of calls, and diverges after n calls. We conjecture that finiteness holds for these modified projections, $\pi^n_{\mathrm{m}+}$.

DEFINITION 5.21 (FINITE PROJECTIONS $\pi^n_{\mathrm{m}+}$)

$$
\begin{array}{l}
\pi^0_{\mathrm{m}-} = \lambda z.\lambda x.\mathtt{bot} \\
\pi^{n+1}_{\mathrm{m}-} = \lambda z.\lambda x. \\
\qquad \mathtt{eif}(\mathtt{is}_1(x), \mathtt{inj}_1(\pi^n_{\mathrm{m}-}(z)(\mathtt{out}_1(x))), \\
\qquad \mathtt{eif}(\mathtt{is}_2(x), \mathtt{inj}_2(\pi^n_{\mathrm{m}-}(z)(\mathtt{out}_2(x))), \ldots, \\
\qquad \mathtt{eif}(\mathtt{is}_k(x), \mathtt{inj}_k(\pi^n_{\mathrm{m}-}(z)(\mathtt{out}_k(x))), \\
\qquad \mathtt{eif}(\mathtt{iscell}(x), \mathtt{set}(w, \mathtt{cons}(x, \mathtt{get}(w))); \\
\qquad\quad \mathtt{eif}(\mathtt{member}(x, z), x, \mathtt{set}(x, \pi^n_{\mathrm{m}-}(\mathtt{cons}(x, z))(\mathtt{get}(x))))), \\
\qquad \mathtt{let}\ c = \mathtt{ref}(n)\ \mathtt{in}\ \lambda x_0. \\
\qquad\quad \mathtt{eif}(\mathtt{iszero}(\mathtt{get}(c)), \mathtt{bot}, \mathtt{set}(c, \mathtt{pred}(\mathtt{get}(c)))); \\
\qquad\quad \mathtt{mapcar}(\lambda x.\pi^n_{\mathrm{m}-}(\mathtt{nil})(x), \mathtt{get}(w)); \\
\qquad\quad \mathtt{let}\ r = \pi^n_{\mathrm{m}-}(\mathtt{nil})(x(\pi^n_{\mathrm{m}-}(\mathtt{nil})(x_0)))\ \mathtt{in} \\
\qquad\quad \mathtt{mapcar}(\lambda x.\pi^n_{\mathrm{m}-}(\mathtt{nil})(x), \mathtt{get}(w));\ r) \ldots)) \\
\pi^n_{\mathrm{m}+} = \mathtt{let}\ w = \mathtt{ref}(\mathtt{nil})\ \mathtt{in}\ \pi^n_{\mathrm{m}-}(\mathtt{nil})
\end{array}
$$

Variable c above is a counter, freshly created for each function projected, to count the number of calls. Besides this one change, these projections are the same as the π^n_{m}.

THEOREM 5.22 Finite Approximation, Theorem 3.14, is provable for the projections $\pi^n_{\mathrm{m}+}$: $\{\pi^n_{\mathrm{m}+}(a) \mid n \in \mathbb{N}\} \cong_{\mathrm{m}} \{a\}$.

PROOF: By Lemma 5.16, it suffices to show

$$\{\pi^n_{\mathrm{m}+}(a) \mid n \in \mathbb{N}\} \cong_{\mathrm{m}} \{\pi^n_{\mathrm{m}}(a) \mid n \in \mathbb{N}\}.$$

Since $\pi^n_{\mathrm{m}+}$ only adds additional possibilities for divergence to π^n_{m}, the $\precsim$ direction is not difficult. For $\succsim$, suppose $C[\pi^n_{\mathrm{m}}(a)]\downarrow$, in m steps; we show $C[\pi^{m+n+1}_{\mathrm{m}}(a)]\downarrow$: in the $C[\pi^n_{\mathrm{m}}(a)]$ computation there can be no more than m application steps, so no single function is applied more than m times, so if the counters for all projected functions are initially set to be larger than m, no counter will ever reach 0. And, π^{m+n+1}_{m} indeed will assure every counter is initially larger than m since termination of $C[\pi^n_{\mathrm{m}}(a)]$ guarantees the minimum projection must be more than π^{m+1}_{m}. □

We conjecture that these finite projections do indeed "project enough" to produce only finitely many programs at each rank.

CONJECTURE 5.23 (FINITENESS) The set

$$\{a \mid a \text{ is closed and } a \cong_{\mathrm{m}} \pi_{+}^{n}(a)\}$$

contains finitely many $\cong_{\mathrm{m}}$-distinct expressions for each $n \in \mathbb{N}$.

In the functional case, extensionality is critical to prove this property: there are finitely many functions at a certain level because by induction, the functions have a finite domain and finite range of elements of the next lowest level, and thus by extensionality there can be only finitely many such functions. In the memory case, the failure of extensionality for closures causes this proof technique to fail. A proof of the conjecture thus appears difficult.

We can at least conclude that there is some hope of developing finite projections in the memory case.

It is also possible to develop a theory of projections for $\mathbb{L}_{\mathrm{m}}$ by adding new, atomic projection operators, following the idea of the $\mathbb{L}_{\mathrm{bpn}\pi-(k)}$ and $\mathbb{L}_{\mathrm{obj}\pi(k)}$ constructions. The atomic projections can be interpreted as the $\mathbb{L}_{\mathrm{bpn}}$ projections via the embedding of Definition 5.3. These atomic projections will have the effect of projecting local memory cells, thus destroying full abstraction.

6 Conclusions

We have studied a fairly broad family of languages, and return with mixed results on whether fully abstract finite element theories may be developed. For functional languages with recognizer operators ($\mathbb{L}_{\mathrm{inj}(k)}$ of Section 3 and $\mathbb{L}_{\mathrm{bpn}(k)}$ of Section 4.1), the prospects are excellent. Without recognizers present ($\mathbb{L}_{\mathrm{bpn}-(k)}$ of Section 4.2), prospects do not look as promising. Simple objects ($\mathbb{L}_{\mathrm{obj}(k)}$ of Section 4.3) apparently have no elegant projection operations π^n definable. For memories, we conjecture that the projection functions π_{+}^{n} yield finitely many expressions at each rank, so the problem is open but there is some possibility of a solution.

We showed it is always possible to develop a theory of projections by adding new, atomic projection operators ($\mathbb{L}_{\mathrm{bpn}\pi-(k)}$ of Section 4.2.1, and $\mathbb{L}_{\mathrm{obj}\pi(k)}$ of Section 4.4). This however may expose internal details and thus lose full abstraction. Lemma 4.16 shows in some cases it is provable that addition of atomic projections changes the equivalence. Many explicit λ-labelling methods will also suffer from the same problem. For some applications, on the other hand, projections of this form may be adequate.

On a positive note, an inductive characterization of fixed points was possible for all languages studied, so for the particular ranked sequence of successively larger approximations to a fixed point, an additional induction principle was gained. For $\mathbb{L}_{\mathrm{obj}}$, an inductive characterization of the self-reference found in objects was also possible. One of the reasons why this characterization was possible across such a wide range of languages is the set ordering $\{\cdot\} \sqsubseteq \{\cdot\}$ applies across the

whole spectrum of languages. This generality is one of the major benefits of using this ordering. Its main weakness is that it does not directly generalize to the nondeterministic case.

Does full abstraction matter? It depends on the problem. In some cases it does not matter, and other times it might be critical. Negative uses of $\underset{\sim}{\sqsubset}$ are the major source of problems when an equivalence that is too fine-grained, $\cong_{\mathrm{too-fine}} \subsetneq \cong$, is used. For instance, extensionality (Theorem 5.8) is one such property. For this reason, extensionality of a more fine-grained equivalence will not imply extensionality of $\cong$. The notion of a faithful ideal (Abadi, Pierce, and Plotkin 1991) also has a negative instance of $\cong$: "if $a \in I$ and $a \cong b$, then $b \in I$". The secondary source of problems is that a more fine-grained equivalence will mean some operational equivalences will not be provable via the too-fine-grained characterization.

Language embeddings also are of interest in their own right. The embeddings studied here give precise lemmas which characterize concepts that informally are well-known, but were without rigorous characterization. The fact that a language with injections alone allowed many other language features to be homomorphically embedded in them gives a case for studying untyped languages of that form. The inability of the pure λ-calculus to serve as the target of any homomorphic embedding defining $\mathbb{L}_{\mathrm{bpn}}$ (Lemma 4.5) justifies why it is often inadequate to study the pure λ-calculus alone as a model of functional programming. The largest divide in language semantics lies between languages which can be homomorphically embedded into simple languages $\mathbb{L}_{\mathrm{inj}(k)}$, and those which have only non-homomorphic embeddings into $\mathbb{L}_{\mathrm{inj}(k)}$. This gap serves as one means to formally separate the functional from non-functional languages. Global effects such as memories lack a homomorphic embedding and are semantically difficult to deal with. Functional objects are homomorphic and justifiably semantically simpler than memories, and in fact the addition of objects may not modify operational equivalence (Conjecture 4.19).

6.1 Other language features

It is at least worth a brief mention of how other language features not discussed may be handled; greater exploration of this topic is a subject for future work. Objects were only partially addressed here. Imperative objects will pose additional difficulties beyond the problems exposed here.

Two features not touched on here are control effects and types. Control effects such as exceptions and `call/cc` are probably manageable, but values that escape to the top will cause complications because values that are not finite may escape: $\pi^4(\texttt{abort}(1000))$ would abort the projection operation and return 1000. If these values are "observable", values of any rank could escape.

Simply-typed higher order functional languages such as PCF are not particularly difficult because type membership may be inductively defined, and given the type of an expression e, its finite projection may be defined statically. More complex

types such as polymorphic, recursive, and parameterized types greatly complicate matters by removing this possibility. In this case, recognizers must exist in the language, but recognizers are difficult to type. So, a `typecase` construct is probably required to preserve full abstraction.

The real question we seek an answer to is whether finite element characterizations are possible for real languages such as Standard ML that combine all of these features, and whether the characterizations may be used to prove deep properties of programs. We are still not quite able to answer that question. One particular challenge is whether it is possible to give a semantics to Standard ML that defines types "semantically," without recourse to type proof systems.

Acknowledgements

The original research on finite projections (Mason, Smith, and Talcott 1996) was done in collaboration with Ian Mason and Carolyn Talcott. The author would like to also thank Carolyn Talcott for a careful reading of an early version of this paper that caught a number of errors; Lemma 5.20 is hers. Laurent Daimi and Andy Pitts also gave careful readings of the paper which the author is thankful for. The author would like to acknowledge support for this work from NSF grants CCR-9301340 and CCR-9312433.

References

Abadi, M., L. Cardelli, and R. Viswanathan (1996). An interpretation of objects and object types. In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*. ACM.

Abadi, M., B. Pierce, and G. Plotkin (1991). Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science 2*(1), 1–21.

Abramsky, S. (1990). The lazy lambda calculus. In *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley.

Agha, G., I. Mason, S. F. Smith, and C. Talcott (1992). Towards a theory of actor computation. In *CONCUR*, Volume 630 of *Lecture notes in Computer Science*, pp. 565–579. Springer-Verlag.

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (Revised ed.), Volume 103 of *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland.

Bloom, B. (1990). Can LCF be topped? *Information and Computation 87*, 264–301.

Egidi, L., F. Honsell, and S. R. della Rocca (1992). Operational, denotational and logical descriptions: a case study. *Fundamenta Informaticae 16*(2), 149–170.

Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico (1995). An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation 8*(4), 357–397.

Felleisen, M. (1991). On the expressive power of programming languages. *Science of Computer Programming 17*, 35–75.

Felleisen, M., D. Friedman, and E. Kohlbecker (1987). A syntactic theory of sequential control. *Theoretical Computer Science 52*, 205–237.

Felleisen, M. and R. Hieb (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 235–271.

Freyd, P., P. Mulry, G. Rosolini, and D. Scott (1990). Extensional PERs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pp. 346–354.

Gordon, A. D. and G. D. Rees (1996). Bisimularity for a first-order calculus of objects with subtyping. In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*.

Harper, R., F. Honsell, and G. Plotkin (1993). A framework for defining logics. *Journal of the Association of Computing Machinery*, 143–184.

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. J. (1996, February). Proving congruence of bisimulation in functional programming languages. *Information and Computation 124*(2), 103–112.

Kamin, S. N. and U. S. Reddy (1994). Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell (Eds.), *Theoretical Aspects of Object-Oriented Programming*, Chapter 13, pp. 464–495. MIT Press.

MacQueen, D. B., G. Plotkin, and R. Sethi (1984). An ideal model of types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*.

Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From operational semantics to domain theory. *Information and Computation 128*(1).

Mason, I. A. and C. L. Talcott (1991). Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 287–327.

Milne, R. E. and C. Strachey (1976). *A theory of programming language semantics*. Chapman and Hall, London, and Wiley, New York.

Milner, R. (1977). Fully abstract models of typed λ-calculi. *Theoretical Computer Science 4*, 1–22.

Mitchell, J. (1993). On abstraction and the expressive power of programming languages. *Science of Computer Programming 21*.

Mosses, P. D. (1992). *Action Semantics*. Cambridge.

Pitts, A. M. (1996, 15 June). Relational properties of domains. *Information and Computation 127*(2), 66–90.

Riecke, J. G. (1993). Fully abstract translations between functional languages. *Mathematical Structures in Computer Science 3*, 387–415.

Ritter, E. and A. M. Pitts (1995). A fully abstract translation between a λ-calculus with reference types and standard ml. In *2nd Int. Conf. on Typed Lambda Calculus and Applications, Edinburgh, 1995*, Volume 902 of *Lecture Notes in Computer Science*, pp. 397–413. Springer-Verlag, Berlin.

Scott, D. (1976). Data types as lattices. *SIAM J. Computing 5*, 522–587.

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.

Talcott, C. L. (1989). Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University, Stanford, CA 94305.

Talcott, C. L. (1997). Reasoning about functions with effects. In A. D. Gordon and A. M. Pitts (Eds.), *Higher Order Operational Techniques in Semantics*. Cambridge University Press. This volume.