Reasoning about Explicit and Implicit Representations of State

Amr Sabry*

Rice University
sabry@cs.rice.edu

John Field

IBM T.J. Watson Research Center
jfield@watson.ibm.com

Abstract

The semantics of imperative languages are often expressed in terms of a store-passing translation and an algebra for reasoning about stores. We axiomatize the semantics of several typical imperative languages via equational axioms by "inverting" the store-passing translation as well as the algebraic axioms for reasoning about the store. The inversion process is simple and systematic and results in theories that are similar to equational theories for imperative languages that have been derived in more complicated ways, and is likely to be applicable to languages other than those presented here.

1 A Semantic Toolkit for Imperative Languages

The mathematical semantics of imperative languages conventionally regards the *store* as a primitive concept [16:p. 203]. Using a "store-passing transformation", imperative operations are translated to *explicit* operations on a concrete representation of the store in an augmented lambda-calculus. The resulting store constructs typically have a simple semantics that is easy to reason about. Furthermore, such a store and its related operations can yield a natural "store algebra" that can be used for optimization purposes [9].

Since it is often the case that one wishes to reason about languages directly rather than by translation, there have been many proposals for logics for imperative languages [2, 4, 6, 7, 8, 10, 11, 17]. Although each of these was designed independently, and differs from the others in subtle ways, the similarities between them suggest the existence of a *systematic* way to derive equational logics from the semantics of the languages.

We propose to construct logics for imperative languages by deriving an *inverse* to the store-passing transformation and applying it to the equational axioms of the enriched lambda-calculus. We explain the general strategy in detail in Section 2. In later sections, we demonstrate the feasibility of our technique by applying it to a variety of imperative languages that are extensions of the call-by-value λ-calculus. The first language includes an infinite set of locations, facilities to assign values to these locations, and the ability to pass them to procedures. The second language extends the first with facilities to perform

*Supported in part by NSF grants CCR 89-17022 and CCR 91-22518 and Texas ATP grant 91-003604014.

dynamic allocation and de-allocation. The last language has a *lazy* store update operation in addition to the conventional strict operation. Although the axioms of the derived equational logics simulate the effect of "passing the store", they represent *local* transformations on the original programs, and do not rely on the concept of a centralized global store.

Our derived logics are more complex than the store-augmented lambda-calculus, reflecting the intricacies of modeling the sequencing of imperative operations. By using the original store-passing transformation as the basis for deriving these logics, they may be obtained more readily than attempting to construct them *ab initio*. We believe this approach is likely to be particularly beneficial for developing logics for languages not designed with equational reasoning in mind.

2 Manufacturing Equational Logics

In this section, we outline the general strategy we will use to axiomatize the semantics of our imperative languages. Let $\mathcal{R}$ be the source language whose semantics we wish to axiomatize. The first step is to define a target language $\mathcal{G}$ with explicit store manipulation constructs and to axiomatize its semantics using an equational theory $\mathrm{ax}(\mathcal{G})$. Fortunately, the derivation of the latter equational theory is usually much simpler than the derivation of a corresponding equational theory for the source language because of the explicit representation of the store.

The semantics of the source language is then defined using a (store-passing) translation $f: \mathcal{R} \rightarrow \mathcal{G}$, which can be used to reason (indirectly) about $\mathcal{R}$-terms using $\mathrm{ax}(\mathcal{G})$. Our goal is to derive a set of axioms $\mathrm{ax}(\mathcal{R})$ with the following property. For all $r_1, r_2 \in \mathcal{R}$,

$$\mathrm{ax}(\mathcal{R}) \vdash r_1 = r_2 \qquad \text{if and only if} \qquad \mathrm{ax}(\mathcal{G}) \vdash f(r_1) = f(r_2).$$

To derive the axioms $\mathrm{ax}(\mathcal{R})$, we develop an *inverse translation*: $f^{-1}: \mathcal{G} \rightarrow \mathcal{R}$, and axioms $A_0^{\mathcal{R}}$ such that for all $r \in \mathcal{R}$:

$$A_0^{\mathcal{R}} \vdash r = (f^{-1} \circ f)(r).$$

The inverse translation f^{-1} must be *compositional*, *i.e.*, for all contexts C and terms g in $\mathcal{G}$,

$$f^{-1}(C[g]) = f^{-1}(C[x])\,[x := f^{-1}(g)]\;,$$

where x does not occur in C. The compositional nature of f^{-1} ensures that the equations resulting from the inversion process are valid in *any* context as required by equational logics in general. Naturally, the axioms $A_0^{\mathcal{R}}$ must be *sound* with respect to the translation f and $\mathrm{ax}(\mathcal{G})$:

$$A_0^{\mathcal{R}} \vdash r_1 = r_2 \quad \text{implies} \quad \mathrm{ax}(\mathcal{G}) \vdash f(r_1) = f(r_2).$$

Next, for each equation $E_i^{\mathcal{G}} \in \mathrm{ax}(\mathcal{G})$ such that: $E_i^{\mathcal{G}} \stackrel{df}{=} (g_1 = g_2)$, find a set of equational axioms $\mathcal{E}_i^{\mathcal{R}}$ such that:

$$\mathcal{E}_i^{\mathcal{R}} \vdash f^{-1}(g_1) = f^{-1}(g_2).$$

If $E_i^{\mathcal{G}}$ is a simple first order equation, then we can define $\mathcal{E}_i^{\mathcal{R}}$ as $\{\, f^{-1}(g_1) = f^{-1}(g_2)\,\}$.

The full equational theory for $\mathcal{R}$ then consists of the axioms $ax(\mathcal{R})$ defined by:

$$\mathrm{ax}(\mathcal{R}) \stackrel{df}{=} A_0^{\mathcal{R}} \cup \bigcup_i \mathcal{E}_i^{\mathcal{R}}.$$

It is straightforward to show that the set $\mathrm{ax}(\mathcal{R})$ proves all the equations that are provable in the target language. In other words, the development results in a complete equational correspondence between the theories for the source and target languages.

This technique is not specific to logics of imperative languages, and is similar to the method used to establish the correspondence between the λ-calculus and combinatory logic [1, 5], and the correspondence between the computational λ-calculus [13] for call-by-value languages and the λ-calculus for continuation-passing terms [15].

3 Strict Assignments, Call-by-Reference

We begin by considering a pure functional language $\mathcal{R}_1$ extended with simple imperative facilities. The functional subset includes values V and applications $(M\ N)$. The set of values includes the natural numbers $\mathbb{N}$, variables, procedures, and a "unit" constant '()' denoting an unspecified value. The language also includes an infinite set of constants A representing locations. Locations may be passed as arguments and may be assigned to using **setref!**. The contents of a location is accessed using the operation **deref**. Although this language is quite simple, it is sufficient to represent, *e.g.*, pointer manipulations and call-by-reference parameter passing, since addresses are computable "first-class" values. The syntax of $\mathcal{R}_1$ is as follows:

Definition 3.1. $(\mathcal{R}_1)$

$$\begin{array}{rcll} M & ::= & V \mid (M\ N) & (\mathit{Terms}) \\ V & ::= & \mathbb{N} \mid X \mid A \mid () \mid (\lambda x.M) \mid \mathsf{deref} \mid \mathsf{setref!} & (\mathit{Values}) \end{array}$$

$$\begin{array}{rcl} X & \in & \mathit{Variables} = \{x, y, z, \ldots\} \\ A & \in & \mathit{Locations} = \{a_1, a_2, \ldots\} \end{array}$$

∎

In order to define the semantics of the language $\mathcal{R}_1$, we introduce the language $\mathcal{G}_1$ that includes simple primitives for manipulating the store. All constructs in $\mathcal{G}_1$ manipulate the store explicitly; in particular, the evaluation of a $\mathcal{G}_1$-program produces a pair that consists of a value and a store. The syntax of $\mathcal{G}_1$ is defined as follows:

Definition 3.2. $(\mathcal{G}_1)$

$$\begin{array}{rcll} P & ::= & \langle W, S\rangle \mid (W\ P) \mid S@W & (\mathit{Terms}) \\ W & ::= & \mathbb{N} \mid X \mid A \mid ? \mid (\lambda\langle x, s\rangle.P) & (\mathit{Values}) \\ S & ::= & s \mid (\mathit{upd}\ S\ W\ W) & (\mathit{Stores}) \end{array}$$

∎

The notation $\langle W, S\rangle$ denotes a pair whose first element is the value W and second element is the store S. The constant '?' refers to an unspecified value. The variable s denotes an unknown store and the term $(\mathit{upd}\ S\ W_1\ W_2)$ represents a store S whose most recent assignment associates expression W_2 with address W_1. Such a store may contain more than one assignment for any given address. The lookup operation $S@W_1$ returns a pair $\langle W_2, S\rangle$ where W_2 is the rightmost, or most recent assignment, to location W_1 in the store S. A $\mathcal{G}_1$-program is a closed term $((\lambda s.P)\ \{\})$ where $\{\}$ is the "initial" empty store.

Given the above informal explanation, we present three axioms that suffice for the evaluation of $\mathcal{G}_1$-programs:

Definition 3.3. $(\mathrm{ax}(\mathcal{G}_1))$

$$\begin{array}{rcll}
((\lambda\langle x, s\rangle.P)\ \langle W, S\rangle) & = & P[x := W, s := S] & (Q_1 = \beta_v^\times) \\
(\mathit{upd}\ S\ A\ W)@A & = & \langle W, (\mathit{upd}\ S\ A\ W)\rangle & (Q_2) \\
(\mathit{upd}\ S\ A_1\ W)@A & = & ((\lambda\langle x, s\rangle.\langle x, (\mathit{upd}\ s\ A_1\ W)\rangle)\ S@A) \quad A_1 \neq A & (Q_3)
\end{array}$$

∎

Axioms Q_1 and Q_2 are straightforward and should be self-explanatory. Axiom Q_3 ensures that store components *not* containing the looked-up address are preserved.

The semantics of the source language $\mathcal{R}_1$ is defined indirectly by the axioms $\mathrm{ax}(\mathcal{G}_1)$, using the following store-passing translation:

Definition 3.4. (*Store-Passing Transformation*)

$$\begin{array}{rcl}
\mathcal{S} : \mathcal{R}_1 \times S & \rightarrow & \mathcal{G}_1 \\
\mathcal{S}[\![V]\!]s & = & \langle \mathcal{W}[\![V]\!], s\rangle \\
\mathcal{S}[\![(M\ N)]\!]s & = & ((\lambda\langle x, s\rangle.(x\ \mathcal{S}[\![N]\!]s))\ \mathcal{S}[\![M]\!]s) \\
 & & \\
\mathcal{W}[\![n]\!] & = & n \\
\mathcal{W}[\![x]\!] & = & x \\
\mathcal{W}[\![()]\!] & = & ? \\
\mathcal{W}[\![(\lambda x.M)]\!] & = & (\lambda\langle x, s\rangle.\mathcal{S}[\![M]\!]s) \\
\mathcal{W}[\![A]\!] & = & A \\
\mathcal{W}[\![\mathsf{deref}]\!] & = & (\lambda\langle x, s\rangle.s@x) \\
\mathcal{W}[\![\mathsf{setref!}]\!] & = & (\lambda\langle x, s\rangle.\langle(\lambda\langle y, s'\rangle.\langle ?, (\mathit{upd}\ s'\ x\ y)\rangle), s\rangle)
\end{array}$$

∎

In the remainder of the section, we will systematically develop an equational theory $\mathrm{ax}(\mathcal{R}_1)$ for $\mathcal{R}_1$. The derived theory will have the property that any equivalence provable *indirectly* via $\mathrm{ax}(\mathcal{G}_1)$ is also provable *directly* in $\mathrm{ax}(\mathcal{R}_1)$. In this sense, $\mathrm{ax}(\mathcal{R}_1)$ will be *complete* relative to $\mathrm{ax}(\mathcal{G}_1)$.

3.1 Inverting the Store-Passing Transformation

The specification of an inverse to the store-passing translation relies on the representation of stores in the source language $\mathcal{R}_1$. We choose the following common representation [2, 12, 14]. A store of the form:

$$(\mathit{upd}\ (\mathit{upd}\ \ldots\ (\mathit{upd}\ (\mathit{upd}\ s\ W_{11}\ W_{12})\ W_{21}\ W_{22})\ \ldots)\ W_{n1}\ W_{n2})$$

will be represented in the source language as the sequence of assignments:

$$()\ ;\ (\mathsf{setref!}\ V_{11}\ V_{12})\ ;\ (\mathsf{setref!}\ V_{21}\ V_{22})\ ;\ \ldots\ ;\ (\mathsf{setref!}\ V_{n1}\ V_{n2}),$$

where V_{ij} corresponds to W_{ij} and the sequencing operation $(M\ ;\ N)$ is an abbreviation for: $((\lambda d.N)\ M)$, where d is not free in N.

The following inverse store-passing transformation defines inverses for arbitrary terms of $\mathcal{G}_1$, and formalizes the store's representation:

Definition 3.5. (*Inverse Store-Passing*)

$$\begin{array}{rcl}
\mathcal{S}^{-1} : \mathcal{G}_1 & \rightarrow & \mathcal{R}_1 \\
\mathcal{S}^{-1}[\![\langle W, S\rangle]\!] & = & \mathcal{I}[\![S]\!]\ ;\ \mathcal{W}^{-1}[\![W]\!] \\
\mathcal{S}^{-1}[\![(W\ P)]\!] & = & (\mathcal{W}^{-1}[\![W]\!]\ \mathcal{S}^{-1}[\![P]\!]) \\
\mathcal{S}^{-1}[\![S@W]\!] & = & \mathcal{I}[\![S]\!]\ ;\ (\mathsf{deref}\ \mathcal{W}^{-1}[\![W]\!]) \\
 & & \\
\mathcal{W}^{-1}[\![n]\!] & = & n \\
\mathcal{W}^{-1}[\![x]\!] & = & x \\
\mathcal{W}^{-1}[\![?]\!] & = & () \\
\mathcal{W}^{-1}[\![(\lambda\langle x, s\rangle.P)]\!] & = & (\lambda x.\mathcal{S}^{-1}[\![P]\!]) \\
\mathcal{W}^{-1}[\![A]\!] & = & A \\
 & & \\
\mathcal{I}[\![s]\!] & = & () \\
\mathcal{I}[\![(\mathit{upd}\ S\ W_1\ W_2)]\!] & = & \mathcal{I}[\![S]\!]\ ;\ (\mathsf{setref!}\ \mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{W}^{-1}[\![W_2]\!])
\end{array}$$

∎

The relationship between the store-passing transformation and its inverse requires the following axioms:

Definition 3.6. ($A_0^{\mathcal{R}_1}$)

$$\begin{array}{rcll}
((\lambda x.M)\ V) & = & M[x := V] & (I_1 = \beta_v) \\
\mathsf{deref} & = & (\lambda x.(\mathsf{deref}\ x)) & (I_2 = \eta_v^d) \\
\mathsf{setref!} & = & (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) & (I_3 = \eta_v^s) \\
(V\ ((\mathsf{setref!}\ V_1\ V_2)\ ;\ M)) & = & (\mathsf{setref!}\ V_1\ V_2)\ ;\ (V\ M) & (I_4) \\
(M\ N) & = & ((\lambda x.(x\ N))\ M) & (I_5)
\end{array}$$

∎

We now prove a tight relationship between the original store-passing transformation and its inverse, using the axioms $A_0^{\mathcal{R}_1}$:

Lemma 3.7 *Let $M \in \mathcal{R}_1$ and S be a store in $\mathcal{G}_1$. Then*

$$A_0^{\mathcal{R}_1} \vdash (\mathcal{I}[\![S]\!]\ ;\ M) = \mathcal{S}^{-1}[\![\mathcal{S}[\![M]\!]S]\!]$$

Proof. The proof is by induction on the structure of M and proceeds by cases:

- $M = V$, then $\mathcal{S}^{-1}[\![\mathcal{S}[\![V]\!]S]\!] = \mathcal{S}^{-1}[\![\langle \mathcal{W}[\![V]\!], S\rangle]\!] = (\mathcal{I}[\![S]\!]\ ;\ \mathcal{W}^{-1}[\![\mathcal{W}[\![V]\!]]\!])$. It remains to show that:
$$A_0^{\mathcal{R}_1} \vdash \mathcal{W}^{-1}[\![\mathcal{W}[\![V]\!]]\!] = V.$$
  - $V = n$ or $V = x$ or $V = A$, then the result is immediate.
  - $V = \lambda x.N$, then:
$$\begin{array}{rcll} \mathcal{W}^{-1}[\![\mathcal{W}[\![(\lambda x.N)]\!]]\!] & = & \mathcal{W}^{-1}[\![(\lambda\langle x, s\rangle.\mathcal{S}[\![N]\!]s)]\!] & \\ & = & (\lambda x.\mathcal{S}^{-1}[\![\mathcal{S}[\![N]\!]s]\!]) & \\ & = & (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ N)) & \text{(induction)} \\ & = & (\lambda x.(()\ ;\ N)) & \\ & = & (\lambda x.N) & (\beta_v) \end{array}$$
  - $V = \mathsf{deref}$, then:
$$\begin{array}{rcl} \mathcal{W}^{-1}[\![\mathcal{W}[\![\mathsf{deref}]\!]]\!] & = & \mathcal{W}^{-1}[\![(\lambda\langle x, s\rangle.s@x)]\!] \\ & = & (\lambda x.\mathcal{S}^{-1}[\![s@x]\!]) \\ & = & (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ (\mathsf{deref}\ x))) \end{array}$$
  - $V = \mathsf{setref!}$, then:
$$\begin{array}{rcl} \mathcal{W}^{-1}[\![\mathcal{W}[\![\mathsf{setref!}]\!]]\!] & = & \mathcal{W}^{-1}[\![(\lambda\langle x, s\rangle.\langle(\lambda\langle y, s'\rangle.\langle ?, (\mathit{upd}\ s'\ x\ y)\rangle), s\rangle)]\!] \\ & = & (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ \lambda y.(\mathcal{I}[\![s']\!]\ ;\ (\mathsf{setref!}\ x\ y)\ ;\ ?))) \end{array}$$
- $M = (N_1\ N_2)$, then:
$$\begin{array}{rcll} \mathcal{S}^{-1}[\![\mathcal{S}[\![(N_1\ N_2)]\!]S]\!] & = & \mathcal{S}^{-1}[\![((\lambda\langle x, s\rangle.(x\ \mathcal{S}[\![N_2]\!]s))\ \mathcal{S}[\![N_1]\!]S)]\!] & \\ & = & ((\lambda x.(x\ (\mathcal{I}[\![s]\!]\ ;\ N_2)))\ (\mathcal{I}[\![S]\!]\ ;\ N_1)) & \\ & = & ((\lambda x.(x\ N_2))\ (\mathcal{I}[\![S]\!]\ ;\ N_1)) & (\beta_v) \\ & = & \mathcal{I}[\![S]\!]\ ;\ ((\lambda x.(x\ N_2))\ N_1) & \\ & & \text{(Repeated application of } (I_4)) & \\ & = & \mathcal{I}[\![S]\!]\ ;\ (N_1\ N_2) & (I_5) \end{array}$$

∎

3.2 Inverting the Store-Passing Axioms

To derive the complete equational theory that corresponds to the axioms $\mathrm{ax}(\mathcal{G}_1)$, we apply the inverse store-passing transformation to these axioms. The result of inverting Q_2 and Q_3 yields the axioms I_6 and I_7, respectively:

Definition 3.8. (I_6 *and* I_7)

$$
\begin{array}{rcll}
(\mathsf{setref!}\ A\ V)\ ;\ (\mathsf{deref}\ A) & = & (\mathsf{setref!}\ A\ V)\ ;\ V & (I_6) \\
(\mathsf{setref!}\ A_1\ V)\ ;\ (\mathsf{deref}\ A) & = & ((\lambda x.((\mathsf{setref!}\ A_1\ V)\ ;\ x))\ (\mathsf{deref}\ A)) \quad A_1 \neq A & (I_7)
\end{array}
$$

∎

The remaining axiom, Q_1, is not a simple first-order equation (it involves substitution). However, it turns out that the equivalence between the inversions of its right and left sides can be established using the set of axioms we have already developed. To show this, we require the following lemmas:

Lemma 3.9 (Substitution) *Let P and W be terms of the appropriate sort in $\mathcal{G}_1$, then $\mathcal{S}^{-1}[\![P]\!][x := \mathcal{W}^{-1}[\![V]\!]]$ is identical to $\mathcal{S}^{-1}[\![P[x := W]]\!]$.*

Proof Idea. The proof is by induction on the structure of P and proceeds by cases. ∎

Lemma 3.10 *Let P, S and S_1 be terms of the appropriate sort in $\mathcal{G}_1$, then:*

- $\mathrm{ax}(\mathcal{R}_1) \vdash (\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!]) = \mathcal{S}^{-1}[\![P[s := S]]\!]$.
- $\mathrm{ax}(\mathcal{R}_1) \vdash (\mathcal{I}[\![S]\!]\ ;\ \mathcal{I}[\![S_1]\!]) = \mathcal{I}[\![S_1[s := S]]\!]$.

Proof. The proof is by induction on the structure of P or S_1. The proof relies on the fact that there is exactly one free store variable s in terms P and S and no free store variables in W:

- $P = \langle W_1, S_1\rangle$, then:

$$
\begin{array}{rcl}
\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!] & = & \mathcal{I}[\![S]\!]\ ;\ \mathcal{I}[\![S_1]\!]\ ;\ \mathcal{W}^{-1}[\![W_1]\!] \\
& = & \mathcal{I}[\![S_1[s := S]]\!]\ ;\ \mathcal{W}^{-1}[\![W_1]\!] \\
& = & \mathcal{S}^{-1}[\![\langle W_1, S_1[s := S]\rangle]\!]
\end{array}
$$

- $P = (W_1\ P_1)$, then:

$$
\begin{array}{rcl}
\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!] & = & \mathcal{I}[\![S]\!]\ ;\ (\mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{S}^{-1}[\![P_1]\!]) \\
& = & (\mathcal{W}^{-1}[\![W_1]\!]\ (\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P_1]\!])) \\
& = & (\mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{S}^{-1}[\![P_1[s := S_1]]\!])
\end{array}
$$

- $P = S_1 @ W_1$, then:

$$
\begin{array}{rcl}
\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!] & = & \mathcal{I}[\![S]\!]\ ;\ \mathcal{I}[\![S_1]\!]\ ;\ (\mathsf{deref}\ \mathcal{W}^{-1}[\![W_1]\!]) \\
& = & \mathcal{I}[\![S_1[s := S]]\!]\ ;\ (\mathsf{deref}\ \mathcal{W}^{-1}[\![W_1]\!]) \\
& = & \mathcal{S}^{-1}[\![S_1[s := S] @ W_1]\!]
\end{array}
$$

- $S_1 = s$, then:

$$\begin{aligned}\mathcal{I}[\![S]\!] \;;\; \mathcal{I}[\![S_1]\!] &= () \;;\; \mathcal{I}[\![S_1]\!] \\ &= \mathcal{I}[\![S_1]\!] \\ &= \mathcal{I}[\![S_1[s := s]]\!]\end{aligned}$$

- $S_1 = (\mathit{upd}\ S_2\ W_1\ W_2)$, then:

$$\begin{aligned}\mathcal{I}[\![S]\!] \;;\; \mathcal{I}[\![S_1]\!] &= \mathcal{I}[\![S]\!] \;;\; \mathcal{I}[\![S_2]\!] \;;\; (\mathsf{setref!}\ \mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{W}^{-1}[\![W_2]\!]) \\ &= \mathcal{I}[\![S_2[s := S]]\!] \;;\; (\mathsf{setref!}\ \mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{W}^{-1}[\![W_2]\!]) \\ &= \mathcal{I}[\![(\mathit{upd}\ S_2[s := S]\ W_1\ W_2)]\!]\end{aligned}$$

∎

Using Lemmas 3.9 and 3.10, we are now in a position to show that axiom Q_1 is invertible:

Lemma 3.11 *Let P, S, and W be terms of the appropriate sort in $\mathcal{G}_1$. Then*

$$(I_1\text{–}I_7) \vdash \mathcal{S}^{-1}[\![((\lambda\langle x, s\rangle.P)\ \langle W, S\rangle)]\!] = \mathcal{S}^{-1}[\![P[x := W, s := S]]\!]$$

Proof. The inverse of the left-hand side of the equation is:

$$\begin{aligned}((\lambda x.\mathcal{S}^{-1}[\![P]\!])\ (\mathcal{I}[\![S]\!] \;;\; \mathcal{W}^{-1}[\![W]\!])) &= \mathcal{I}[\![S]\!] \;;\; ((\lambda x.\mathcal{S}^{-1}[\![P]\!])\ \mathcal{W}^{-1}[\![W]\!]) \\ &\quad \text{(Repeated application of } (I_4)) \\ &= \mathcal{I}[\![S]\!] \;;\; \mathcal{S}^{-1}[\![P]\!][x := \mathcal{W}^{-1}[\![W]\!]] \qquad (\beta_v) \\ &= \mathcal{I}[\![S]\!] \;;\; \mathcal{S}^{-1}[\![P[x := W]]\!] \\ &\quad \text{(Substitution Lemma 3.9)} \\ &= \mathcal{S}^{-1}[\![P[x := W, s := S]]\!] \\ &\quad \text{(Lemma 3.10)}\end{aligned}$$

∎

The complete set of axioms $\mathrm{ax}(\mathcal{R}_1)$ in Figure 1 consists of all the previously derived axioms. It proves the same equations that $\mathrm{ax}(\mathcal{G}_1)$ proves on store-passing terms:

Theorem 3.12 $\mathrm{ax}(\mathcal{R}_1) \vdash M = N$ *if and only if* $\mathrm{ax}(\mathcal{G}_1) \vdash \mathcal{S}[\![M]\!]s = \mathcal{S}[\![N]\!]s$ *(where s is a free variable).*

Proof. Each direction is by induction on the length of proofs, using Lemmas 3.7 and 3.11, the derived axioms (I_6) and (I_7), and the fact that $\mathcal{S}^{-1}[\![\cdot]\!]$ is compositional. ∎

4 Dynamic Allocation

We now extend the store-passing language with an operation that binds locations. The resulting language $\mathcal{G}_2$ is more realistic than $\mathcal{G}_1$, since it can express various storage allocation (and de-allocation) protocols.

$$
\begin{array}{rcllr}
((\lambda x.M)\ V) & = & M[x := V] & & (I_1 = \beta_v) \\
\mathsf{deref} & = & (\lambda x.(\mathsf{deref}\ x)) & & (I_2) \\
\mathsf{setref!} & = & (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) & & (I_3) \\
(V\ ((\mathsf{setref!}\ V_1\ V_2)\ ;\ M)) & = & (\mathsf{setref!}\ V_1\ V_2)\ ;\ (V\ M) & & (I_4) \\
(M\ N) & = & ((\lambda x.(x\ N))\ M) & & (I_5) \\
(\mathsf{setref!}\ A\ V)\ ;\ (\mathsf{deref}\ A) & = & (\mathsf{setref!}\ A\ V)\ ;\ V & & (I_6) \\
(\mathsf{setref!}\ A_1\ V)\ ;\ (\mathsf{deref}\ A) & = & ((\lambda x.((\mathsf{setref!}\ A_1\ V)\ ;\ x))\ (\mathsf{deref}\ A)) & A_1 \neq A & (I_7)
\end{array}
$$

Figure 1: Complete Set of Axioms $\mathrm{ax}(\mathcal{R}_1)$.

Definition 4.1. ($\mathcal{G}_2$) The language $\mathcal{G}_2$ extends $\mathcal{G}_1$ as follows:

$$P \ ::= \quad \ldots \mid (alloc\ A\ P)$$

∎

In the term $(alloc\ A\ P)$ the location A is *bound* in P. The usual notion of free variables in a term P is extended to include locations bound by *alloc*, and is notated $FV(P)$. The formal semantics of the new construct is the following:

$$
\begin{array}{rcllr}
(W\ (alloc\ A\ P)) & = & (alloc\ A\ (W\ P)) & A \notin FV(W) & (Q_4) \\
(alloc\ A\ P) & = & P & A \notin FV(P) & (Q_5)
\end{array}
$$

The first axiom uses an idea of Felleisen *et al.* [8]; it "lifts" the binding of the location A to permit the evaluation of the application $(W\ P)$. The condition $A \notin FV(W)$ ensures that *alloc* does not capture any free locations A in W and may require that the bound location A be renamed, *e.g.*,

$$((\lambda\langle d, s\rangle.s@A)\ (alloc\ A\ \langle A, S\rangle)) = (alloc\ A_1\ ((\lambda\langle d, s\rangle.s@A)\ \langle A_1, S\rangle)).$$

The second axiom is a garbage collection rule for de-allocating references. The theory $\mathrm{ax}(\mathcal{G}_2)$ includes all the axioms in $\mathrm{ax}(\mathcal{G}_1)$ as well as Q_4 and Q_5.[1]

Since the store-passing language $\mathcal{G}_2$ is more expressive than the previous language $\mathcal{G}_1$, we can derive an equational theory for a correspondingly more expressive imperative language:

Definition 4.2. ($\mathcal{R}_2$) The syntactic definition of the language $\mathcal{R}_2$ extends the one for $\mathcal{R}_1$ with the following clause:

$$V \ ::= \ \ldots \mid (\mathsf{letref}\ A\ M)$$

∎

[1]Our approach to dynamic allocation has the advantage of simplicity and abstractness, but relies on the meta-operation of renaming to ensure uniqueness of addresses [8]. A more "concrete" allocation mechanism could be defined in any number of ways, but would require more complicated axioms to correctly reflect its internal behavior.

The semantics of the new construct is as expected:

$$\mathcal{S}[\![(\mathsf{letref}\ A\ M)]\!]s \;=\; (alloc\ A\ \mathcal{S}[\![M]\!]s)$$

Informally, letref creates a reference cell and binds it to a new name.[2]

Using the inverse store-passing translation for *alloc*:

$$\mathcal{S}^{-1}[\![(alloc\ A\ P)]\!] \;=\; (\mathsf{letref}\ A\ \mathcal{S}^{-1}[\![P]\!])$$

we can derive the equational theory for $\mathcal{R}_2$ that proves all the equations that $\mathcal{G}_2$ proves on store-passing terms. The development is similar to the one in the previous section. The resulting set of axioms includes all the axioms $ax(\mathcal{R}_1)$ and the following additional axioms:

$$\begin{array}{rcllr}
(\mathsf{letref}\ A\ (V\ M)) & = & (V\ (\mathsf{letref}\ A\ M)) & A \notin FV(V) & (I_8) \\
(\mathsf{setref!}\ V_1\ V_2)\ ;\ (\mathsf{letref}\ A\ M) & = & (\mathsf{letref}\ A\ ((\mathsf{setref}\ V_1\ V_2)\ ;\ M)) & A \notin FV(V_1, V_2) & (I_9) \\
(\mathsf{letref}\ A\ M) & = & M & A \notin FV(M) & (I_{10})
\end{array}$$

4.1 Related Calculi

When compared to other systems, the $\mathcal{R}_2$-calculus has several distinguishing features. First, in contrast with calculi derived from abstract machines for the source imperative language [4, 8, 11], our equational theory never eliminates an expression of the form (setref! V_1 V_2) (cf. Section 5). These expressions simply migrate from point to point within the program to allow lookup operations. In the other calculi [4, 8, 11], there is typically a distinguished position in the program text that contains the most recent value associated with a particular location: setref! expressions modify this current value and lookup operations copy this current value.

The $\mathcal{R}_2$-calculus is closer in spirit to the theory $\boldsymbol{\lambda}_{var}$ [14] and the Imperative Lambda Calculus (ILC) [17] but differs in several technical respects. The theory $\boldsymbol{\lambda}_{var}$ uses a call-by-name semantics for procedures and has an additional **pure** construct. It also uses a "monadic" operator to express the sequencing of assignments while the sequencing of assignments in $\mathcal{R}_2$ is implicit in the order of evaluation of applications. Finally, ILC does not include "first-class" reference cells as procedures cannot create reference cells and return them, *e.g.*, the term (letref A A) is a valid program in our language but corresponds to a "dangling" pointer in ILC.

5 Extensions

Previous sections have concentrated on deriving calculi that are sufficiently powerful to evaluate programs. In this section, we turn our attention to extended target calculi and their effect on logics for source languages. We will concentrate on adding axioms for proving equivalences on open terms that are useful for optimization. In addition, we will consider certain functional extensions.

[2] The language can express ML-like reference cells as follows:

$$\mathsf{ref} \;=\; \lambda x.(\mathsf{letref}\ A\ ((\mathsf{setref!}\ A\ x)\ ;\ A)).$$

5.1 Lazy Store

We extend $\mathcal{G}_2$ with a *lazy* store update operation in addition to the conventional strict operation:

$$S \ ::= \ \ldots \mid (upd_\ell \ W \ P)$$

The lazy store semantics generalizes the strict semantics [3] and can be used to justify many optimizations in imperative languages that would be inconsistent with strict store semantics [3, 9]. Intuitively, the evaluation of a program may terminate in the lazy store semantics but diverge in the strict store semantics. However, if both evaluations terminate then the two semantics agree on the final answer. The formal semantics of the new lazy store update operation is defined in terms of the old update operation as follows:

$$(upd_\ell \ A_1 \ P)@A \ = \ ((\lambda\langle x, s\rangle.(upd \ s \ A_1 \ x)@A) \ P)$$

In other words, the evaluation of P does not happen until requested by a lookup operation.

By following a development similar to the previous sections, we extend the source language with a corresponding lazy assignment:

$$M \ ::= \ \ldots \mid (\mathsf{setref!}_\ell \ M \ M)$$

The translations between the two languages are:

$$\mathcal{S}[\![(\mathsf{setref!}_\ell \ M_1 \ M_2)]\!]s \ = \ ((\lambda\langle x, s'\rangle.\langle ?, (upd_\ell \ x \ \mathcal{S}[\![M_2]\!]s')\rangle) \ \mathcal{S}[\![M_1]\!]s)$$

$$\mathcal{I}[\![(upd_\ell \ W \ P)]\!] \ = \ (\mathsf{setref!}_\ell \ \mathcal{W}^{-1}[\![W]\!] \ \mathcal{S}^{-1}[\![P]\!])$$

This definition implies that the *location* into which a value is stored is computed strictly, but the expression associated with the location is not evaluated until it is needed by some dereference operation.

The resulting theory for the source language includes all the previously derived axioms and the following axioms:

$$\begin{aligned}
(\mathsf{setref!}_\ell \ A_1 \ M) \ ; \ (\mathsf{deref} \ A) &= ((\lambda x.((\mathsf{setref!} \ A_1 \ x) \ ; \ (\mathsf{deref} \ A))) \ M) && (I_{11}) \\
(\mathsf{setref!} \ V_1 \ V_2) \ ; \ (\mathsf{setref!}_\ell \ V \ M) &= (\mathsf{setref!}_\ell \ V \ ((\mathsf{setref!} \ V_1 \ V_2) \ ; \ M)) && (I_{12}) \\
(\mathsf{setref!}_\ell \ M_1 \ M_2) &= ((\lambda x.(\mathsf{setref!}_\ell \ x \ M_2) \ ; \ ?) \ M_1) && (I_{13})
\end{aligned}$$

5.2 Extensions to the Imperative Fragment

We can extend the axioms for the imperative fragments to include equations that axiomatize the store datatype. This is the approach taken in the logic PIM [9], and we consider a few similar extensions here. For example, we may include the following axioms:

$$\begin{aligned}
(upd \ (upd \ S \ A_1 \ W_1) \ A_2 \ W_2) &= (upd \ (upd \ S \ A_2 \ W_2) \ A_1 \ W_1) \\
&\quad\ A_1 \neq A_2 \text{ and } A_1 \notin FV(W_2) \\
(upd \ (upd \ S \ A \ W_1) \ A \ W_2) &= (upd \ S \ A \ W_2)
\end{aligned}$$

The corresponding axioms in the source language are immediately obtained by applying the inverse store-passing transformation:

$$\begin{aligned}
(\textsf{setref!}\ A_1\ V_1)\ ;\ (\textsf{setref!}\ A_2\ V_2) &= (\textsf{setref!}\ A_2\ V_2)\ ;\ (\textsf{setref!}\ A_1\ V_1) \\
&\ A_1 \neq A_2 \text{ and } A_1 \notin FV(V_2) \\
(\textsf{setref!}\ A\ V_1)\ ;\ (\textsf{setref!}\ A\ V_2) &= (\textsf{setref!}\ A\ V_2)
\end{aligned}$$

5.3 Extensions to the Functional Fragment

We can also extend the axioms for the functional fragments to a more powerful call-by-value logic such as the computational λ-calculus [13]. Because the sequencing operation can be expressed using procedures, extensions to the functional fragment prove more equations between imperative terms. The following example shows the effect of adding the axiom β_{lift} [13, 15]:

$$\begin{aligned}
&(\textsf{setref!}_\ell\ A\ M)\ ;\ (\textsf{deref}\ A)\ ;\ (\textsf{deref}\ A_1) \\
&\quad = ((\lambda x.(\textsf{setref!}\ A\ x)\ ;\ (\textsf{deref}\ A))\ M)\ ;\ (\textsf{deref}\ A_1) && (I_{11}) \\
&\quad = ((\lambda x.(\textsf{setref!}\ A\ x)\ ;\ x)\ M)\ ;\ (\textsf{deref}\ A_1) && (I_5) \\
&\quad = ((\lambda x.(\textsf{setref!}\ A\ x)\ ;\ x\ ;\ (\textsf{deref}\ A_1))\ M) && (\beta_{lift}) \\
&\quad = ((\lambda x.(\textsf{setref!}\ A\ x)\ ;\ (\textsf{deref}\ A_1))\ M) && (\beta_v) \\
&\quad = (\textsf{setref!}_\ell\ A\ M)\ ;\ (\textsf{deref}\ A_1) && (I_{12})
\end{aligned}$$

The equation is not provable without the additional axiom.

5.4 Extensions to the Store Representation

In addition to providing additional axioms to obtain more complete logics, it is also possible to provide *functional* extensions to our store algebras to model additional source language constructs. For example, arrays can be modeled in a straightforward manner by locations that are tuples of the form $\langle$*base*, *offset*$\rangle$, where *base* is one of the "atomic" locations we have defined previously, and *offset* is an integer offset. The location comparison implicitly required by the store lookup axioms must be correspondingly redefined to operate on such pairs. Implicit renaming performed by the *alloc* construct carries over without change to the extended representation for locations.

6 Assessment

We have shown here that equational logics for a family of representative programming languages can be defined in a particularly effective way by translation to simple, "orthogonal" target calculi. These calculi extend the untyped lambda-calculus with store constructs and simple store algebras.

Imperative programs typically have two aspects that complicate reasoning: the inherent behavior of the store, and the complications of sequencing operations on the store amidst other fundamental operations. When the store *sequencing* aspects of the language are encoded by translation to a target calculus, our approach allows source axioms to be developed that mirror the behavior of the translation, independently of the axioms that merely "implement the store."

Although our derived source axioms are complete *relative* to the axioms of the store algebra, the latter theories do not provide a complete axiomatization of the observational equivalence relation for any of the presented imperative languages, and are therefore *semantically* incomplete. It is expected that we cannot achieve semantic completeness with untyped languages such as the ones we used in the previous sections. However, the incompleteness problem is much more fundamental. Even in a simply typed language, assignments can be used to define recursive or cyclic data structures, *e.g.*, the evaluation of the following simply typed program diverges:[3]

$$(\mathsf{setref!}\ A^{ref(int\to int)}\ (\lambda x^{int}.x^{int}))\ ;\ (\mathsf{setref!}\ A\ (\lambda x.((\mathsf{deref}\ A)\ x)))\ ;\ ((\mathsf{deref}\ A)\ 0).$$

The technique of deriving equational logics by analyzing the translations between two languages is necessarily limited by the "expressive power" of the source language. In other words, for each term in one language, there should be a corresponding term in the other language that has the same behavior in all contexts. Cases of mismatch between the constructs in the languages require special consideration. Indeed, it need not be the case that *any* equational logic exists, however derived, which can prove all equations that are true in some semantic model—even when considering only closed terms.

However even when equational logics cannot be derived by our technique, we believe there is considerable benefit to adopting our approach to defining the semantics of imperative languages via translation, since this process may be used to obtain deeper understanding of the language's semantics.

Acknowledgments

We thank Matthias Felleisen for suggesting several improvements in the presentation and for clarifying several historical facts. We also thank the reviewers for helpful comments.

References

[1] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, revised edition (1984).

[2] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985) 637–655.

[3] Cartwright, R. and Felleisen, M. The semantics of program dependence. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1989) 13–27.

[4] Crank, E. and Felleisen, M. Parameter-passing and the lambda calculus. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages* (1991) 233–244.

[3]It is possible to use a type system that rejects the above program and that guarantees that the evaluation of "simply typed" programs terminates [17]. With such restrictions, it may be possible to axiomatize the observational equivalence relation.

[5] Curry, H. B. and Feys, R. *Combinatory Logic, Volume I.* North-Holland, Amsterdam (1958).

[6] Demers, A. and Donahue, J. Making variables abstract: An equational theory for Russell. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages* (1983) 59–72.

[7] Felleisen, M. and Friedman, D.P. A calculus for assignments in higher-order languages. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages* (1987) 314–325.

[8] Felleisen, M. and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102 (1992) 235–271. Technical Report 89-100, Rice University.

[9] Field, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis (preliminary report). In *ACM Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report RR-909, Yale University (1992) 98–107.

[10] Hoare, C., Hayes, I., Jifeng, H., Morgan, C., Roscoe, A., Sanders, J., Sorensen, I., Spivey, J., and Sufrin, B. Laws of programming. *Communications of the ACM*, 30, 8 (1987) 672–686.

[11] Mason, I. and Talcott, C. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1, 3 (July 1991) 287–327.

[12] Mason, I.A. and Talcott, C. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105, 2 (1992) 167–215. Preliminary version in Proceedings of the 4th IEEE Symposium on Logic in Computer Science 1989.

[13] Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science* (1989) 14–23. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.

[14] Odersky, M., Rabin, D., and Hudak, P. Call by name, assignment, and the lambda calculus. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (January 1993) 43–56.

[15] Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Functional Programming* (1993). To appear. Preliminary version in Proceedings of the ACM Conference on Lisp and Functional Programming, 1992. Also Technical Report 92-180, Rice University.

[16] Strachey, C. Towards a formal semantics. In Steel, T.B., editor, *Formal Language Description Languages for Computer Programming*, North-Holland (1966) 197–220.

[17] Swarup, V., Reddy, U., and Ireland, E. Assignments for applicative languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1991) 192–214.