Refining First-Class Stores

J. Gregory. Morrisett
Carnegie Mellon University
jgmorris@cs.cmu.edu

March 1993

Abstract

A *first-class store* is an object that captures the values of mutable objects at a particular time in a program's execution. First-class stores allow programmers to cleanly, safely, and efficiently implement "undo" and "redo" operations on mutable objects. This paper describes a generalized interface for first-class stores that allows the programmer to *partition* mutable data and refine the scope of store objects. We demonstrate the power of the interface by discussing three applications: version arrays, replay debugging, and nested transactions. A denotational semantics for first-class stores is given and two implementations are discussed.

1 Introduction

First-class continuations provide programmers with the ability to manipulate the control of their program in many flexible ways. Operationally, we can think of capturing the current continuation as capturing the current program counter and lexical environment of a program and binding it to a variable. Similarly, invoking a continuation installs a previously captured program counter and lexical environment.

Using continuations, programs can "travel in time" with respect to the current program counter and lexical environment. Being able to back up in time is useful for implementing "aborts" of subprograms. However, continuations only capture *part* of the state of a program. In particular, continuations do not capture the state of mutable objects.

This paper describes language extensions which allow the programmer to capture and restore the state of mutable objects in a program. The extensions add two constructs, *store names* and *first-class stores*. Store names provide the ability to *partition* mutable objects into logically disjoint regions. First-class stores are snapshots of values of mutable objects associated with a store name. Previous work concerning first-class stores ([7, 13]) only allows the capture of *all* mutable objects. Partitioning the store appears to be crucial to applications which require the ability to capture and restore state.

We demonstrate the utility of the extensions through three applications: version arrays, a replay debugger, and a nested-transaction system. We sketch the semantics of a simple ML-like language extended with first-class stores and store names and show how these extensions can be implemented within Standard ML (SML). Finally, and most importantly,

we show that the extensions can be integrated with the compiler and runtime of a language to produce an efficient realization of the primitives.

We assume that the reader is familiar with Standard ML.

2 Informal Description

Figure 1 gives an SML signature for first-class stores. The signature defines four new types (`store_name`, `'a rref`, `'a rarray`, `captured_store`) and associated operations. This section gives an informal description of these types and operations and shows how they are used. Section 4 clarifies the semantics of the operations.

A `store_name` is used as a handle to group mutable data into a logically distinct store. This allows us to capture and restore *portions* of the program's mutable state. Some applications, like the debugger and transaction system described in Section 3, require at least two stores: One store is used to group objects whose state is never captured or restored. Usually, these objects are internal to whatever abstraction the application is providing. The other store is used for data whose state might be captured or restored. But some applications, notably the version array example of Section 3.1, require a dynamic number of stores.

A `'_a rref` (restorable ref) is a mutable object that is associated with some store. A `'_a rref` is created by applying `rref` to a `store_name` and a `'_a` value. The `rref` can be read/written using the `get`/`set` operations respectively. As we shall see, there are times when the value contained in an `rref` is undefined. In such a case, if the object is read, the exception `Undefined` is raised. We also provide restorable arrays (`rarray`) with analogous operations and semantics.

A `captured_store` represents the state of the mutable objects associated with a particular store at some point in time. The function `current_store` conceptually returns a snapshot of the store associated with the `store_name` argument. The function `restore_store` restores a store to the state it was in when it was captured. The function `store_name_of` returns the `store_name` that a captured store is a snapshot of.

3 Applications

Leeman [9] gives an excellent overview of applications of "undo" and "redo", including editors, debuggers, interactive systems, etc. Undo and redo seem to be important to a variety of distributed systems applications as well, including Time Warp-style simulation [6] and optimistic concurrency control [8]. This section describes three applications of first-class stores as an undo/redo mechanism in depth: version arrays, replay debugging, and a nested transaction system.

3.1 Version Arrays

Version arrays are functional arrays which provide essentially constant time access and update operations on the most recent "version" of the array [4, 5, 1]. Figure 2 gives an SML signature for version arrays. Programs that use version arrays have the benefit of referential transparency with the added benefit of an expected efficient running time.

```
signature STORE =
  sig
      type store_name
      val new_store_name : unit -> store_name

      exception Undefined

      type 'a rref
      val rref : store_name * '_a -> '_a rref
      val get : 'a rref -> 'a
      val set : 'a rref * 'a -> unit

      type 'a rarray
      val rarray : store_name * int * '_a -> '_a rarray
      val rsub : 'a rarray * int -> 'a
      val rupdate : 'a rarray * int * 'a -> unit

      type captured_store
      val current_store : store_name -> captured_store
      val restore_store : captured_store -> unit
      val store_name_of : captured_store -> store_name
  end
```

Figure 1: First-class Store Signature

```
signature VERSION_ARRAY =
  sig
      type 'a v_array

      val v_array : int * '_a -> '_a v_array
      val sub_v : 'a v_array * int -> 'a
      val update_v : 'a v_array * int * 'a -> 'a v_array
  end
```

Figure 2: Version Array Signature

```
functor Version_Array (structure Store : STORE) : VERSION_ARRAY =
  struct
     local
         open Store
     in
         type 'a v_array = 'a rarray * captured_store

         fun v_array (size,elt) =
             let val sn = new_store_name ()
                 val a = rarray(sn,size,elt)
                 val cs = current_store sn
             in
                 (a, cs)
             end

         fun sub_v ((a,cs),i) = (restore_store cs; rsub(a,i))

         fun update_v ((a,cs),i,x) =
               (restore_store cs;
                rupdate (a,i,x);
                (a,current_store (store_name_of cs)))
     end
  end
```

Figure 3: Version Array Implementation

One way to implement version arrays is to actually do destructive updates on the array, and then do something "special" if an older version is accessed. Figure 3 shows how version arrays may be implemented in SML extended with first-class stores. In this code, a version array is represented as a recoverable (mutable) array and a captured store. When a version array is created, a new store name is allocated, the mutable array is created and associated with this store name, and then the current store with respect to the store name is captured. The recoverable array and captured store make up the initial version of the object. Any time a version is accessed, we first restore the captured store associated with the version. On an update operation, we mutate the recoverable array, and capture a new version of the store.

Note that the ability to partition version arrays into separate store regions is critical to implement this abstraction. Also note that this application requires that the user be able to both "undo" and "redo" side effects.

The above is an acceptable implementation only if the `restore_store`, `current_store`, `rsub`, and `rupdate` operations execute in constant time when used to access the latest version of an array. In Section 5, we show that this is possible.

3.2 Replay Debugging

Tolmach [12] describes a source-level replay debugger for Standard ML of New Jersey (SML/NJ) [2]. The debugger allows the user to run the program forward or backward in time, set breakpoints, and examine bindings. The debugger works by inserting debug-

ging hooks into the source code before compilation. This approach avoids problems of portability and mapping low-level representations back to the source-level.

Being able to run the program *backward* in time allows one to find the point where a program "goes wrong". In addition, the debugger uses backward time travel internally to implement certain features such as identifier lookup and location-based break-pointing.

Tolmach simulates reverse execution by using a "checkpoint and roll-forward" technique. The basic idea is to checkpoint the entire state of the program periodically. If the debugger wants to go back to a specific point in the program's execution, it restores the latest checkpoint that occurred before that point and rolls the program forward to the breakpoint. Tolmach uses SML/NJ's continuations to checkpoint the environment and program counter. However, he has no language mechanism to checkpoint mutable data.

One possibility is to record the values of all mutable objects at a checkpoint. However, SML programs tend to do mutation infrequently. Consequently, Tolmach inserts code to record mutations in a log. To roll back to a checkpoint, he "undoes" the effects recorded in the log. It is interesting to note that to implement the logging efficiently "requires unsafe type coercions and some support from the garbage collector" [12, page 75].

First-class stores provide precisely the abstraction needed to checkpoint the store of the program. The debugger would use one store name `s`, to group all of the mutable objects of the program to be debugged. Any mutable objects internal to the debugger would be associated with another store name `internal_s`. When the debugger wanted to take a checkpoint, it would capture the program's current continuation and use the `current_store` operation to capture the current values of the mutable objects associated with `s`.

3.3 Nested Transactions

The Venari project [14] proposes to add a persistent heap, threads, and transactions to SML/NJ. Nettles and Wing [10] have shown that the implementation of single-threaded transactions can be broken into two separate facilities: Persist and Undo. The Undo facility provides the means to checkpoint the entire store and the ability to restore the last checkpoint. Checkpoints are not exported as first-class objects, because there is never a need in the transaction system, to "abort an abort".

The Venari Undo signature is shown in Figure 4. Two operations are provided, `checkpoint` and `restore`. The `checkpoint` operation takes a "thunk" and checkpoints the entire store of the program before invoking the thunk. The `restore` operation restores the last checkpoint and raises the exception `Restore`, passing it the argument exception. [1]

Venari's Undo facility is trivial to implement with first-class stores. Figure 5 shows how this might be done. It assumes that programmers associate all of their mutable data with the `global_store` exported by the implementation. As with the debugger, an internal store name, `internal_store` is used to group the mutable data (`store_stack`) that is internal to the application.

[1] The exception mechanism is used to checkpoint the control and environment of the program.

```
signature UNDO =
  sig
      exception Restore of exn
      val checkpoint : (unit -> 'a) -> 'a
      val restore : exn -> 'a
  end
```

Figure 4: Venari Undo Signature

```
functor Undo (structure Store : STORE) :
          sig
            include UNDO
            type store_name
            val global_store : store_name
          end =
  struct
     local
         open Store
         val internal_store = new_store_name ()
         val store_stack : captured_store list rref =
                             rref(internal_store,[])
         fun push s = set(store_stack, s :: (get store_stack))
         exception Pop
         fun pop () =
             case (get store_stack) of
               [] => raise Pop
             | (hd::tl) => (set(store_stack,tl); hd)
     in
         type store_name = Store.store_name
         val global_store = new_store_name ()

         exception Restore of exn

         fun checkpoint f =
           let val s = current_store global_store
               val _ = push s
               val result = f ()
           in
               pop ();
               result
           end

         fun restore e = (restore_store (pop ()); raise Restore e)
     end
  end
```

Figure 5: Implementing Undo

$$
\begin{array}{rrcrcl}
(\textit{values}) & d & \in & \mathbf{D} & = & \mathbf{1} + \mathbf{F} + (\mathbf{D} \times \mathbf{D}) + \mathbf{R} + \mathbf{CS} + \mathbf{N} \\
(\textit{functions}) & f & \in & \mathbf{F} & = & ((\mathbf{D} \times \mathbf{G}) \to (\mathbf{D} \times \mathbf{G}))_\perp \\
(\textit{global states}) & g & \in & \mathbf{G} & = & \mathbf{PS} \times \mathcal{P}_{\mathit{fin}}(\mathbf{L}) \times \mathcal{P}_{\mathit{fin}}(\mathbf{N}) \\
(\textit{ref cells}) & r & \in & \mathbf{R} & = & \mathbf{N} \times \mathbf{L} \\
(\textit{program stores}) & \sigma & \in & \mathbf{PS} & = & \mathbf{R} \to_{\mathit{fin}} \mathbf{D} \\
(\textit{captured stores}) & (n, \sigma) & \in & \mathbf{CS} & = & \mathbf{N} \times \mathbf{PS} \\
(\textit{environments}) & \rho & \in & \mathbf{Env} & = & \mathit{Ident} \to_{\mathit{fin}} \mathbf{D} \\
(\textit{locations}) & l & \in & \mathbf{L} & & \\
(\textit{store names}) & n & \in & \mathbf{N} & & \\
 & L & \in & \mathcal{P}_{\mathit{fin}}(\mathbf{L}) & & \\
 & N & \in & \mathcal{P}_{\mathit{fin}}(\mathbf{N}) & &
\end{array}
$$

Figure 6: Semantic Domains

4 Semantics

This section defines the meaning of the first-class store operations when added to a simple functional language. The abstract syntax for our language is:

$$E \quad ::= \quad () \mid x \mid C \mid E_1 E_2 \mid \lambda x.E \mid (E_1, E_2)$$

$$C \quad ::= \quad \texttt{rref} \mid \texttt{get} \mid \texttt{set} \mid \texttt{new_store} \mid \texttt{current_store} \mid \texttt{restore_store} \mid \texttt{store_name_of}$$

where x ranges over a set of identifiers, *Ident*.

Our semantic domains are shown in Figure 6. The changes from standard domains are few and simple: Denotable values ($\mathbf{D}$) are extended to include captured stores ($\mathbf{CS}$) and store names ($\mathbf{N}$). Ref cells ($\mathbf{R}$) have a store name ($\mathbf{N}$) component in addition to a location ($\mathbf{L}$) component. Global states ($\mathbf{G}$) include a program store ($\mathbf{PS}$) that maps locations to values, and sets of locations and sets of store names. The latter two components are used to generate unique locations and store names respectively. We assume that locations ($\mathbf{L}$) and store names ($\mathbf{N}$) are simple algebraic domains, equipped with an equality relation.

Figure 7 gives a denotational semantics for E. Projections are represented by SML-style pattern matching. $[\alpha \mapsto \beta]\gamma$ denotes the function γ extended such that α maps to β. We use $\mathbf{1}$ to denote the single element of the domain $\mathbf{1}$.

Four functions are defined: $\mathcal{E}$ is a function from expressions, environments, and global states to values and global states. $\mathcal{E}$ is a completely conventional semantics of a higher-order call-by-value language with global, single-threaded state. Consequently, we ignore details regarding injections, error values, and non-termination. Note also, that $\mathcal{E}$ can be easily replaced with a continuation semantics. $\mathcal{C}$ is a partial function from constant primitives operations (C), values, and global states to values and global states. The meaning of the first-class store operations are defined here. The functions *new_location* and *new_store_name* are used to generate new locations and store names respectively.

When `current_store` is applied to a store name n, we simply tag the current program store σ with n and return the pair as a captured store (`CS`) value. When `restore_store` is applied to a captured store (n, σ'), we compose σ' with the current program store σ to

$$
\begin{array}{rcl}
\mathcal{E} & : & E \to \mathbf{Env} \to \mathbf{G} \to (\mathbf{D} \times \mathbf{G}) \\
\mathcal{C} & : & C \to \mathbf{D} \to \mathbf{G} \to (\mathbf{D} \times \mathbf{G}) \\
\mathit{new_location} & : & \mathcal{P}_{\mathit{fin}}(\mathbf{L}) \to (\mathbf{L} \times \mathcal{P}_{\mathit{fin}}(\mathbf{L})) \\
\mathit{new_store_name} & : & \mathcal{P}_{\mathit{fin}}(\mathbf{N}) \to (\mathbf{N} \times \mathcal{P}_{\mathit{fin}}(\mathbf{N}))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![()]\!]\,\rho\,g & = & (\mathbf{1}, g) \\
\mathcal{E}[\![x]\!]\,\rho\,g & = & (\rho\,x, g) \\
\mathcal{E}[\![C]\!]\,\rho\,g & = & (\lambda(d,g').(\mathcal{C}[\![C]\!]\,d\,g'), g) \\
\mathcal{E}[\![E_1 E_2]\!]\,\rho\,g & = & \mathit{let}\,(f,g') = \mathcal{E}[\![E_1]\!]\,\rho\,g \\
 & & \mathit{in}\ \mathit{let}\,(d,g'') = \mathcal{E}[\![E_2]\!]\,\rho\,g'\ \mathit{in}\ f(d,g'') \\
\mathcal{E}[\![\lambda x.E]\!]\,\rho\,g & = & (\lambda(d,g').(\mathcal{E}[\![E]\!]\,([x \mapsto e]\rho)\,g'), g) \\
\mathcal{E}[\![(E_1, E_2)]\!]\,\rho\,g & = & \mathit{let}\,(d_1,g') = \mathcal{E}[\![E_1]\!]\,\rho\,g \\
 & & \mathit{in}\ \mathit{let}\,(d_2,g'') = \mathcal{E}[\![E_2]\!]\,\rho\,g'\ \mathit{in}\ ((d_1,d_2),g'')
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}[\![\texttt{rref}]\!]\,(n,d)\,(\sigma,L,N) & = & \mathit{let}\,(l,L') = \mathit{new_location}(L) \\
 & & \mathit{in}\ \mathit{let}\ r = (n,l)\ \mathit{in}\ (r, ([r \mapsto d]\sigma, L', N)) \\
\mathcal{C}[\![\texttt{get}]\!]\,r\,(\sigma,L,N) & = & (\sigma\,r, (\sigma,L,N)) \\
\mathcal{C}[\![\texttt{set}]\!]\,(r,d)\,(\sigma,L,N) & = & (\mathbf{1}, ([r \mapsto d]\sigma, L, N)) \\
\mathcal{C}[\![\texttt{new_store_name}]\!]\,\mathbf{1}\,(\sigma,L,N) & = & \mathit{let}\,(n,N') = \mathit{new_store_name}(N)\ \mathit{in}\ (n, (\sigma,L,N')) \\
\mathcal{C}[\![\texttt{current_store}]\!]\,n\ (\sigma,L,N) & = & ((n,\sigma), (\sigma,L,N)) \\
\mathcal{C}[\![\texttt{restore_store}]\!]\,(n,\sigma')\,(\sigma,L,N) & = & (\mathbf{1}, (\lambda(n',l).\mathit{if}\ n' = n\ \mathit{then}\ \sigma'(n',l)\ \mathit{else}\ \sigma(n',l), L, N)) \\
\mathcal{C}[\![\texttt{store_name_of}]\!]\,(n,\sigma')\,(\sigma,L,N) & = & (n, (\sigma,L,N))
\end{array}
$$

$$
\mathit{new_location}\,L \;=\; (l, L \cup \{l\}) \quad \mathit{where}\ l \notin L
$$

$$
\mathit{new_store_name}\,N \;=\; (n, N \cup \{n\}) \quad \mathit{where}\ n \notin N
$$

Figure 7: Semantic Equations

$$\mathcal{C}[\![\texttt{restore_store}]\!](n,\sigma')(\sigma,L,N) \;=\; (\mathbf{1},(\lambda(n',l).\mathit{if}\ n'=n\ \mathit{and}\ (n',l)\in \mathit{dom}(\sigma')\ \mathit{then}\ \sigma'(n',l)\ \mathit{else}\ \sigma(n',l),L,N))$$

Figure 8: Alternative Semantics

produce a new program store. The composition is done such that the resulting store will map a ref cell $r = (n',l)$ to d if (a) $n' = n$ and $\sigma'(r) = d$, or else (b) $n' \neq n$ and $\sigma(r) = d$.

An alternate semantics for `restore_store`, as shown in Figure 8, is to compose σ' and σ such that the resulting store will map r to d if (a) $n' = n$ and $\sigma'(r) = d$ or else (b) r is not in the domain of σ' and $\sigma(r) = d$.

The difference between the two semantics is subtle, but important. A `rref` cell always has a value under the alternative semantics but can be undefined for the original. To see this, consider the following expression where we use $(\mathit{let}\, x = E_1\ \mathit{in}\ E_2)$ as an abbreviation for $((\lambda x.E_2)E_1)$:

$$\begin{array}{l}\mathit{let}\ n = \texttt{new_store}()\\ \mathit{in}\ \mathit{let}\ s = \texttt{current_store}\, n\\ \quad \mathit{in}\ \mathit{let}\ r = \texttt{rref}(n,())\\ \quad\quad \mathit{in}\ \mathit{let}\ x = \texttt{restore_store}\, s\\ \quad\quad\quad \mathit{in}\ \texttt{get}\, r\end{array}$$

Under the semantics shown in Figure 7, starting with an empty environment, store, etc., the capture of n will yield an empty map. Thus, when s is restored, the resulting program store will be the empty map. Consequently, the meaning of the subexpression `get` r and thus the entire expression is undefined. Under the alternative semantics of Figure 8, this expression denotes an answer with a value component of $\mathbf{1}$.

5 Implementation

In this section, we show how first-class stores can be implemented reasonably efficiently within (portable) SML. The implementation is related to efficient version array techniques [1]. We then describe an implementation that provides a single store, but takes advantage of SML/NJ's garbage collector to give an even more efficient realization of the `current_store` and `restore_store` primitives. We sketch how the implementation can be modified to accommodate multiple stores.

5.1 Implementation in SML

Figure 9 shows how the `STORE` signature of Figure 1 (minus the recoverable arrays) can be implemented within SML. The basic idea is to keep track of changes that are made when moving from one store to the next, instead of actually check-pointing the values of mutable objects.

```
structure Store : STORE = struct
    exception Undefined
    datatype captured_store =
      CS of {store_name  : store_name,           ident : unit ref,
             parent      : captured_store option, depth : int,
             undo        : (unit -> unit),        redo  : (unit -> unit)}
    withtype store_name =
      {parent : captured_store option, depth : int,
       undo   : (unit -> unit),        redo  : (unit -> unit)} ref
    type 'a rref = (store_name * ('a option ref))

    fun id x = x
    exception Check
    fun check (SOME s) = s
      | check NONE = raise Check

    fun new_store_name () = ref {parent=NONE, depth=0, undo=id, redo=id}

    fun store_name_of (CS {store_name=sn,...}) = sn

    fun current_store (sn as ref {parent=p,depth=d,undo=u,redo=r}) =
       let val cs = CS {store_name=sn,parent=p,depth=d,undo=u,redo=r,ident=ref()}
       in
           sn := {parent=SOME cs,depth=d+1,undo=id,redo=id}; cs
       end

    fun move_store (st1 as (CS s1), st2 as (CS s2)) =
       if (#ident s1 = #ident s2) then ()
       else if (#depth s2 < #depth s1) then
           ((#undo s1) (); move_store (check (#parent s1), st2))
       else (move_store (st1, check (#parent s2)); (#redo s2) ())

    fun install (cs as (CS {store_name=sn,parent=p,depth=d,undo=u,redo=r,ident})) =
       sn := {parent=SOME cs,depth=d+1,undo=id,redo=id}

    fun restore_store (cs as (CS {store_name=sn,...})) =
       (move_store (current_store sn, cs); install cs)

    fun get (_,r) = (check (!r)) handle Check => raise Undefined

    fun set ((s,x),v') =
       let val {parent=p,depth=d,undo=u,redo=r} = !s
           val v = !x
           val u' = u o (fn () => x := v)
           val r' = (fn () => x := SOME v') o r
       in
           x := SOME v'; s := {parent=p,depth=d,undo=u',redo=r'}
       end

    fun rref (sn,v) =
       let val x = ref NONE
       in
           set ((sn,x),v);
           (sn,x)
       end
end (* Store *)
```

Figure 9: First-Class Store Implementation

Captured store values for a given store are organized into a tree. The nodes of the tree represent captured stores, and the edges represent side-effects that took place in between the capture of the parent and the child. Restoring a captured store can be accomplished by moving along the edges of the tree, undoing effects as we move up the tree, and re-doing effects as we move down. Obviously, it is only necessary to move up to the least-common ancestor (LCA) of the current captured store and the target captured store. A parent will have multiple children only if it is restored and then some side effects are done to mutable objects associated with the store. In the time travel analogy, subtrees of a node represent "parallel universes".

A `captured_store` is represented as a record of six components: The `store_name` field points to the `store_name` of the captured store. The `ident` field is used for equality testing. The `parent` field points to the parent captured store of this node (if any). The `depth` field is used to calculate the LCA of two captured stores. The `undo` field is a function which undoes all side effects that occurred in between the capture of the parent and this captured store. The `redo` field is a function which re-does all side effects that occurred in between the capture of the parent and this captured store. We use functions to represent the undo and redo information instead of sets of `rref` cells and values, since there is no type-safe way in SML to directly represent such a heterogeneous set.

A `store_name` is a mutable pointer to a record of four components whose purpose is completely analogous to their corresponding components in a `captured_store` record. When a side effect occurs (e.g., `set`), the `undo` and `redo` fields of the record are updated appropriately.

An `'a rref` value is represented as a tuple consisting of a `store_name` and an SML `'a option ref` cell. When a `rref` cell is created, its `ref` component is initialized to `NONE` and then a `set` is used to record the initial side effect. If a `get` operation is performed on a cell whose `ref` component holds `NONE`, the exception `Undefined` is raised. This organization faithfully implements the semantics described by Figure 7 of Section 4.

It is possible to implement the alternative semantics of Figure 8 by keeping time-stamps associated with captured stores and ref cells. The undo/redo functions would undo/redo their effects only if the `rref` cell was created before the target captured store was captured. Since `rref` cells always have values under this semantics, we can eliminate the check in the `get` operation and consequently, the extra level of indirection.

While this implementation is reasonably efficient, it does not allow all inaccessible captured stores to be garbage collected until the entire store (including mutable objects associated with the store) can be collected. To see this, note that an accessible captured store will have a pointer to its parent, whether the parent is needed to move to another accessible store or not. The following section describes an implementation which is integrated with a garbage collector in order to solve this problem.

5.2 Integrated Implementation

Some language implementations already require logging of changes to some mutable values. For instance, a generational garbage collector requires information regarding pointers from older generations to younger generations. Since older data can point to younger data only if a side effect occurs, it is sufficient to record in a log mutations to ref cells in all but the youngest generations. The SML/NJ garbage collector actually records almost all mutated

ref cells in a store list. In fact, the compiler generates code to record the mutations if it cannot statically determine that the update is a non-pointer value. This strategy is reasonable since SML programs do little mutation and this avoids checking to see if a mutable object lives in the youngest generation.

We have modified the SML/NJ compiler and runtime system to provide an efficient realization of first-class stores. The implementation is similar to the one described in the previous section, but takes advantage of SML/NJ's store list and garbage collector. The current implementation provides only a single (implicit) store name.

The key idea behind this implementation is to allow the garbage collector to collect captured stores (and their undo/redo data) that are no longer needed. From the algorithm shown in Figure 9, it is apparent that only captured store nodes along the paths from the reachable captured stores to their least-common ancestor are needed.

Only a few changes were required in the compiler:

- We added `captured_store` as a primitive type to the compiler. The representation is similar to the one described in the previous section. However, the undo and redo information are encoded as a single (`rref` $\times$ value) list (see below).
- We modified the mutation operations (i.e., `:=` and `update`) to *always* record the modified location and also the previous value.

The changes to the runtime system were a bit more elaborate. First, we modified the garbage collector in the following ways:

- The collector was changed to process the new store list. As entries are found, they are moved to a list of entries associated with the single, global store.
- The collector was modified to record all reachable captured stores. Parent field pointers of captured store nodes are ignored by the collector.
- After the initial pass of the collector, the LCA of the set of reachable captured stores is determined. Nodes that are not along a path from a reachable captured store to the LCA are freed.
- The collector then processes each of the saved stores, being sure to preserve any undo data that they point to.

Next, we implemented the operations on `captured_store` values:

- `current_store` makes a call to the garbage collector to do a minor collection. After the collection is complete, all entries from the store list have been moved to the global store name. A new `store` record is allocated as in the previous section, and the store name is reset.
- `restore_store` makes a call to `current_store`, receiving the source store. It computes the least-common ancestor of the source and target store. While computing the least-common ancestor, it undoes the effects listed in its undo list. As the effects are undone, a redo list is calculated. The redo list then takes the place of the undo list. Finally, the effects along the path from the least-common ancestor to the target store are redone. The redo information has been encoded in the undo lists of these nodes.

Many improvements can be made to this simple implementation. To eliminate the need for a minor collection when capturing a store, the mutation operations can record the undo entry directly in the store record and then add a pointer to this entry to the store list. This will slow down mutation slightly, but make capturing and restoring stores efficient. For applications such as the debugger, mutation might be done much more frequently than store capture/restore and the current implementation is preferable. However, for applications such as version arrays, capture/restore will happen quite frequently.

The undo information recorded in a store can be optimized, as well. When processing the undo information for a store, we need to keep around only the first and last entries, since the other entries will never have a visible effect. Log entries for objects that are no longer reachable can be removed as well. Tolmach describes efficient techniques for doing this [12, page 97].

Generalizing this implementation to provide multiple stores is not difficult and we are currently working on this. We plan to associate SML `ref` and `array` values with a single (implicit) store. However, we will export this store and provide explicit `rref` and `rarray` creators which will allow the user to associate a mutable item with a particular store.

The following changes must be made to the compiler to accommodate multiple stores:

- Add `store_name` as a primitive type to the compiler. The representation would be similar to the one shown in Figure 9. Add the `new_store_name` function and an initial global store name.
- Add a `store_name` pointer to all mutable objects and to all `captured_store` objects. Update the creation functions of mutable objects appropriately. Add the `rref`, `rarray`, and `store_name_of` functions.
- Modify the collector to move store list entries to the appropriate `store_name` record instead of the single global store name. Modify `current_store` and `restore_store` to work with the appropriate `store_name` record instead of the single global store name.

6 Related Work and Conclusions

We are not the first to propose first-class stores as a language construct. Johnson and Duggan [7] describe a language, GL, which provides continuations, partial-continuations, and first-class stores. These features are used primarily to implement meta-level tools such as debugging. They give a denotational description of their language and describe an implementation for first-class stores based on the persistent data structures of Driscoll et al. [3]. However, they do not provide any means for partitioning the store. The debugging and transaction applications of Section 3 required at least two store names while the version array application requires an unlimited number of store names.

Wilson and Moher [13] suggest a "call with captured store" (`callcs`) primitive that captures both the current continuation and the current store as a single first-class object. Unfortunately, such a construct has limited use since the user must restore both the continuation and the captured store. Thus, `callcs` cannot be used to implement version arrays.

The implementation of the Venari project's Undo mechanism for SML/NJ was the basis and inspiration of this work [10]. In particular, the need to undo only part of the mutable data

motivated store names. In his thesis, Tolmach briefly explores the idea of using first-class stores for check-pointing [12]. In addition, he explores generalized undo/redo facilities for input/output – necessary components for debugging.

We have provided a generalization of the first-class store mechanism, given a denotational description, and provided prototype implementations. Much work still remains: We would like to relate our implementations to the semantics and formally prove that our optimizations are correct. We are examining other potential applications including Time Warp style event simulation. A particularly challenging problem is to integrate first-class stores with a concurrent language such as CML [11].

7 Acknowledgements

Thanks to Andrzej Filinski for comments on this paper and helping me with the semantic concepts. Thanks to Scott Nettles for his implementation guidance. Thanks to Jeannette Wing for proof reading.

References

[1] A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28:490–503, 1988.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag.

[3] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth ACM Symposium on Theory of Computing*, pages 109–121, May 1986.

[4] S. Holmström. How to handle large data structures in functional languages. In *Proc. of the SERC/Chalmers Workshop on Declarative Programming, University College, London*, 1983.

[5] J. Hughes. An efficient implementation of purely functional arrays. Technical report, Dept. of Computer Sciences, Chalmers University of Technology, Göteborg, 1985.

[6] D. R. Jefferson. Virtual time. *ACM Trans. Prog. Lang. Syst.*, 7(3):404–425, July 1985.

[7] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego*, pages 158–168, Jan. 1988.

[8] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[9] G. B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Prog. Lang. Syst.*, 8(1):50–87, Jan. 1986.

[10] S. M. Nettles and J. M. Wing. Persistence + undoability = transactions. In *Proceedings of the Hawaii International Conference on Systems Science 25*, Jan. 1992. Also available as Technical Report CMU-CS-91-173, School of Computer Science, Carnegie Mellon University, August 1991.

[11] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[12] A. P. Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, Oct. 1992. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-378-92.

[13] P. R. Wilson and T. G. Moher. Demonic memory for process histories. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989. Also published as SIGPLAN Notices 24(7), July 1989.

[14] J. M. Wing. The Venari project: Goals and plans. Venari Note 1, School of Computer Science, Carnegie Mellon University, 1990, internal note.