Persistence + Undoability = Transactions

Scott M. Nettles and Jeannette M. Wing

August 30, 1991

CMU-CS-91-173

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
Keywords: persistence, recoverability, transactions, databases, Standard ML
Abstract

Persistence means objects live potentially forever. Undoability means that any change to a program's store can potentially be undone. In our design and implementation of support for single-threaded nested transactions in Standard ML of New Jersey (SML/NJ), we provide persistence and undoability as orthogonal features and combine them in a simple and elegant manner.

We provide support for persistence through an SML interface that lets users manipulate a set of persistent roots and provides a save function that causes all data reachable from the persistent roots to be moved into the persistent heap. We implement the interface through simple extensions to SML's generational garbage collector and maintain the persistent heap using CMU's Recoverable Virtual Memory system.

We provide support for undoability through an SML interface that exports two functions: checkpoint, which checkpoints the current store, and restore, which undoes all changes made to the previously checkpointed store. The implementation takes advantage of the simple runtime representation of data in SML and, as for persistence, extends the existing garbage collector scheme. SML's "mostly" functional nature allows us to implement this abstraction without undue performance penalty.

Finally, we combine these capabilities to support single-threaded nested transactions by defining a higher-order function transact that guarantees the permanence of effects of committed transactions. We succinctly define transact completely in terms of the interfaces for persistence and undoability. Unlike other transaction-based programming languages like Argus or Avalon/C++, we need not add new control structures; moreover, we handle aborts of nested or top-level transactions using SML's exception mechanism.
1. Motivation

1.1. Revisiting Transactions

Transactions are a well-known and fundamental control abstraction that arose out of the database community. A transaction is a group of operations that is performed atomically ("all-or-nothing"). Traditional database applications like electronic banking and airline reservations systems rely on properties of transactions to guarantee the consistency of the data they read and modify. Systems such as Tabs [S*85] and Camelot [EMS91] demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus [LS83] and Avalon/C++ [DHW88] go one step further by providing linguistic support for transactions in the context of a general-purpose programming language. In principle programmers can now use transactions as a unit of encapsulation to structure an application program without regard for how they are implemented at the operating system level.

In practice, however, transactions have yet to be shown useful in general-purpose applications programming. The problem is a mismatch between what applications need and what transactions provide. State-of-the art transactional facilities provide support for distributed, concurrent, nested transactions in a completely integrated operating system layer or programming language. These facilities were built with database applications like electronic banking in mind. Hence, they were designed and tuned for that application domain, where typically short-lived transactions operate on large-sized objects. However, the concept of a transaction is useful in its own right, not just for database applications. Some applications, such as object repositories and the Coda highly available file system [S*90], need support for single-site, non-nested, single-threaded transactions that access small, simple objects for short time periods measured in milliseconds. Other applications, such as CAD/CAM and software development environments [HN86, LPRS88], need support for transactions that access (and usually infrequently update) large, complex data structures for long time periods measured in hours or days. Builders of these applications have the choice of buying in toto an integrated transactional facility tuned for performance characteristics different from the applications' or building from scratch a facility with the same functionality but tailored specifically for their performance needs. These applications would like to exploit the transaction abstraction but current transactional facilities treat them as anomalous cases.

In this paper we revisit support for transactions by adopting a "pick-and-choose" approach rather than a "kit-and-kaboodle" approach. We provide separate modules to support different transactional properties individually and then compose these modules to provide transactional semantics. To illustrate our approach in detail we will focus on single-site, single-threaded nested transactions. In this context we can view the persistence and undoability properties of transactions as completely orthogonal. In Section 6 we will discuss how we expect to build upon this work to handle distributed, concurrent, multi-threaded transactions.

Our approach keeps support for separate properties separable and modular; as a result, our design is simple and elegant. Of course, we do not avoid the inherent semantic complexity of transactions, borne by its non-trivial model of computation [Win89, Wei89, LM86], but we provide users with more flexibility to choose what guarantees they need for their application.
1.2. Why SML?

We cast our approach concretely in the context of programming languages. Instead of designing a brand new language from scratch, we target an existing language as a basis for extension. For technical and practical reasons, we chose Standard ML of New Jersey as our base language. Henceforth we will use SML to mean just the language and SML/NJ to mean the New Jersey implementation of SML. SML is a strongly-typed, mostly functional, programming language. At its core, it supports functions as first-class values, exceptions, and polymorphism. SML's modules facility supports information hiding, data abstraction, and parameterized modules. Most notably, SML has a published formal semantics [MTH90], which means that any extension has the potential of being formally defined and can be objectively evaluated in terms of how much it perturbs the existing semantics. One important practical reason for choosing SML as our base language is that a decent compiler and runtime were readily available and relatively easy to extend. Another practical reason is that SML has a growing local (CMU) and international user community. Finally, we chose to target the New Jersey implementation of SML because SML/NJ supports continuations ¹ and it runs on different architectural and operating system platforms.

In the design and implementation of our own extensions, we gain additional leverage from SML's high-level language features and SML/NJ's well-modularized design. SML makes a type distinction between immutable and mutable values (refs); we rely on strong typing to let the runtime system safely operate on addresses (without the programmer's knowledge). We use signatures to separate interface information from implementation and functors to compose parameterized modules. We exploit SML/NJ's highly-phased compiler by not modifying its front-end at all. We modify its back-end with small additions that fit neatly into its garbage collection scheme and take advantage of its simple runtime representation of data; we use the storage allocation algorithm unchanged.

We assume some familiarity of SML and explain details of examples as necessary, especially our use of SML's modules facility.

1.3. Example

As a running example, we use the relation abstraction whose signature is given in Figure 1. We can obviously use relations to implement a relational database [Dat77].

Create constructs a new relation from a given a set of attributes. Insert (delete) returns a new relation that is the result of adding (removing) a given tuple into a given relation. An tuple is a set of bindings between attributes and values. For illustrative reasons, we also choose to have both insert and delete modify their relation argument. Both raise the exception InvalidTuple if the number of values given in the tuple argument is not the same as the number of attributes in the relation.

Union, intersect, difference, and product are pure (side-effect free) functions that perform the usual set operations on relations. They each raise the exception InvalidAttributes if the sets of attributes for the two relation arguments are not the same. Select returns a new relation that contains all tuples that satisfy a given predicate (the boolean functional argument). We show the interfaces for the set and relational database operations for completeness only.

¹SML as defined in [MTH90] does not feature continuations, but see [DHM91] for a formal description.
signature RELATION = sig
  type relation (* A set of attributes to which values can be assigned. *)
  type rtuple (* A set of bindings between attributes and values. *)
  type attributes

  (* Constructors *)
  val create: attributes -> relation
  exception InvalidRTuple
  val insert: rtuple * relation -> relation
  val delete: rtuple * relation -> relation

  (* Basic set operations *)
  exception InvalidAttributes
  val union: relation * relation -> relation
  val intersect: relation * relation -> relation
  val difference: relation * relation -> relation
  val product: relation * relation -> relation

  (* A basic relational database operation *)
  val select: (rtuple -> bool) -> relation -> relation
end

Figure 1: Signature for Relations

signature BIND_RELATION = sig
  structure Relation : RELATION
  type identifier

  val bind: (Relation.relation * identifier) -> unit
  val unbind: identifier -> unit

  exception UnboundId
  val fetch: identifier -> Relation.relation
end

Figure 2: Signature for Bindable Relations

Their implementations have no effect on our discussion of persistence and undoability.

Bindable relations (Figure 2) extends relations by adding bind, unbind, and fetch functions. Bind lets us bind to an identifier an entire relation; unbind has the side effect of disassociating the relation bound to a given identifier; fetch returns the relation bound to its identifier argument or raises an exception if the identifier is unbound.

In the SML modules facility, a structure is a kind of module that implements the interface specified in a signature. A functor is a parameterized module that, when instantiated, creates a structure. Hence, large, modular SML programs typically consist of signatures and functors. Programmers create structures by functor application, which is analogous to instantiation of a parameterized module in many other programming languages like Ada [Dep83] or CLU [L*.81].

For our example, we assume there are two functors: Relation(): RELATION, which takes no parameters and returns a structure that matches the RELATION signature; and Bind_Relation( Relation: RELATION): BOUND_RELATION, which takes as a parameter a structure that matches the RELATION signature and returns a structure that matches the BOUND_RELATION signature. Below, we use these two functors, first to create a relation structure, Relation, and next
to create a bindable relation structure, Bind_Relation, by functor application on the structure Relation:\footnote{Since structure names and functor names are in disjoint namespaces, we follow the standard SML naming convention: the structure named on the left-hand side of the equal symbol has the same name as the functor applied on the right-hand side.}

\begin{verbatim}
structure Relation = Relation();
structure Bind_Relation = Bind_Relation(Relation);
\end{verbatim}

In the next three sections we extend the two signatures given in Figures 1 and 2 to support persistent relations (Section 2), "undoable" relations (Section 3), and finally transactional relations (Section 4). For each section, we first explain informally the model of computation, give the design of our extension, give details of our implementation, and illustrate a use of the extension on the relation example, reusing the Relation structure created above. In Section 5 we describe our current implementation status and present some preliminary performance results. We close this paper in Section 6 with a discussion of related and future work.

2. Persistence

An object that is persistent is one that outlives the computation that created it. Persistent objects live potentially forever. In our current design for SML, any first-class value can be a persistent object. Formally, any member in the semantic domain \textit{Val} can be made persistent. \footnote{See p. 47 in [MTH90] for a detailed definition of \textit{Val}.}

2.1. Model of Computation

Informally, here are the modifications and additions we make to the dynamic semantics of SML:

- We add to the domain of values, \textit{Val}, a new domain of persistent memory addresses, \textit{PAddr}.

- We add the notion of a persistent memory, \textit{PMem}: \textit{PAddr} \rightarrow \textit{Val}, a finite mapping from persistent addresses to values. Persistent memory co-exists with the usual SML memory (bindings between "normal" addresses and values).

- We add the notion of a persistent environment, \textit{PEnv}, which co-exists with the usual SML environment (bindings between identifiers and values). \textit{PEnv} can be thought of as a symbol table containing bindings between identifiers and values. In particular, a persistent address can be bound to an identifier, thus giving us a way to access the persistent memory through the persistent environment. Conceptually, the persistent environment contains a set of persistent "roots" into persistent memory.

2.2. Interface

The interface to the persistent memory and persistent environment is shown in the signature in Figure 3. Before explaining each function in detail, consider the following typical scenario for using them. At startup, an SML user links to the persistent environment through a call to \textit{init}. The user can choose to add to and remove entries from the
signature PERS = sig
  exception InitFailed
  val init: string * string * bool -> unit

  exception SaveFailed
  val save: unit -> unit

  val bind: identifier * 'a -> unit
  val unbind: identifier -> unit

  exception UnboundId
  val retrieve: identifier -> 'a
end

Figure 3: Signature for Persistence

persistent environment through bind and unbind. The user calls save to save changes made to objects (in persistent memory) reachable from the root set contained in the persistent environment.

More specifically, init has the effect of obtaining a pointer, which we call the persistent handle, to the persistent environment. If its boolean argument is false, the handle points to a new, empty persistent environment (and memory); otherwise, the handle points to a previously saved environment. Its two string arguments are filenames: the first names the log file; the second, the data file. They are needed for the underlying recoverable virtual memory (RVM) system that we use (see Section 2.3) to implement persistent storage. Save has the effect of writing to disk all changes (including additions) to the persistent memory and persistent environment since the last save. Both init and save may raise an exception because of rare I/O problems encountered by RVM.

Bind adds to the persistent environment a binding between an identifier and value. Unbind removes a binding from the persistent environment given an identifier. Retrieve returns the value bound to an identifier in the persistent environment and raises an exception if no binding for the identifier exists. Notice here a need for dynamic types [ACPP91], which SML does not currently support. SML cannot statically determine whether the type of the value returned by a retrieve of some identifier is the same as the type of the value when it was initially bound through a bind.

Our design maintains the principle of orthogonality between persistence and type [ABC+83]: persistence is not a property associated with a type. We also maintain the principle of referential transparency [MA90]: the persistent value retrieved is the same, not a copy, of the value saved and its internal topology is preserved.

In short, our design, which may change as we gain experience with our implementation, provides a single-level of indirection to persistent memory through a "symbol table" of identifier/value bindings. This design decision reflects a compromise between not providing the user with any mechanism at all for naming values to be saved in persistent storage, e.g., by having at most a single persistent root, and forcing the user to always explicitly move, upon each access or modification, values to and from persistent storage by name, e.g., by providing make-persistent/make-volatile operations [CLNW90]. Our approach, which is similar to that taken in other languages and systems like Poly/ML [Mat87], Galileo [ACO85], and Staple [DM90], gives programmers some control over naming and managing persistent values. It also lets us implement persistent storage management efficiently.
2.3. Implementation

2.3.1. SML Veneer

In our implementation we represent persistent memory as part of a persistent heap and the persistent environment as a symbol table that is itself stored in the persistent heap. The persistent heap lives alongside SML/NJ's volatile heap.

We implement the interface for persistence through a thin veneer of SML code, which calls two C routines in SML/NJ's runtime. One routine initializes the persistent heap and returns a pointer, i.e., the persistent handle, to the persistent symbol table; one implements the effects of the save function. We give details of implementing init and save in the next section. Bind, unbind, and retrieve are standard insert, remove, and lookup operations on symbol tables and need no further discussion.

2.3.2. C-level Interface to RVM

We do not directly rely on the standard (Unix) file system to provide actual permanence of effects; instead we use the CMU Recoverable Virtual Memory (RVM) system [MS91] that provides a different abstraction of permanent storage. RVM allows applications to map recoverable unstructured byte arrays, called segments, into a program's address space. RVM supports multi-threaded, non-nested transactions on these segments; i.e., it guarantees the permanence of changes to segments across system crashes by supporting both undo and recovery on segments. For our implementation of persistence for SML, we use only RVM's recoverability features. To ensure changes made to a segment are saved permanently on disk, first we need to inform RVM which locations have been changed, and we need to call RVM's commit operation to force the changes to disk. RVM uses a log to make this force efficient.

Implementing init

We use two RVM segments to implement the persistent heap. The first segment is of fixed size and is location-independent. It contains three pointers, one to the beginning of the heap, one to the end, and one to the location of the persistent symbol table. The first two pointers determine the domain of persistent addresses (PAddr). The third pointer is the persistent handle. The second segment contains the persistent heap (i.e., the actual data area). It is not of fixed size, but it is location-dependent. Upon initialization, we first try to allocate this segment at the previous start of the persistent heap. If unsuccessful, we map it to some free area, and then readjust all the pointers contained in the persistent heap to reflect its new location. When the mapping of the persistent heap into RVM is done, we return the location of the persistent symbol table. We treat this persistent handle as an implicit argument to the save, bind, unbind, and retrieve functions.

Implementing save

The key idea behind implementing save is to garbage collect a set of pointers that point into the persistent heap. SML/NJ's runtime system uses a store list to support a straightforward generational garbage collection algorithm.

---

4RVM itself represents recoverable segments by either Unix raw disk partitions or Unix files.
functor PRelation (Relation : RELATION) : RELATION = struct
...

  fun insert (tup, rel) =
    (Relation.insert (tup, rel);
     Pers.save();
     rel)

  fun delete (tup, rel) =
    (Relation.delete (tup, rel);
     Pers.save();
     rel)

end

Figure 4: Persistent Relations

[App89]. This list records every store to a location that might contain a pointer; it is discarded after every minor collection. We extend the store list to include non-pointer mutations and, at each minor collection, we record any entries that point inside the persistent heap.

Upon a call to save, we first do a minor collection, thereby leaving only one volatile heap. We then do two things: First, for all the items on the store list, we inform RVM that their locations have changed, allowing RVM to log these changes to disk. Second, we consider all items on this list that are pointers to be roots for garbage collection. This garbage collection step copies objects from the volatile heap onto the end of the persistent heap. Once it is done, we update the end-of-heap pointer, and tell RVM to log all the new objects. Finally, we adjust any pointers that point to objects that have been copied out of the volatile heap to point to their respective copies in the persistent heap. When save finishes we have established the property that no pointers exist from the persistent to the volatile heap. (There may, of course, be pointers within each heap and from the volatile to the persistent heap).

2.3.3. Garbage Collecting the Persistent Heap.

We use a simple stop-and-copy garbage collection scheme for the persistent heap. When collection is done, RVM replaces the entire data segment on disk with the new copy. Though collecting the persistent heap incurs a large disk write, we expect it to be an infrequent activity. Further experimental work may suggest a need for optimizing garbage collection of recoverable storage, e.g., using concurrency [Det90].

2.4. Use

To show a sample use of the interface for persistence, consider making our relations persistent (see Figure 4) by extending our previous signature. For persistent relations, we need only modify the insert and delete functions by simply adding a call to Pers.save after we call the insert (delete) function on regular relations.

To show how we manipulate the persistent environment, we define a functor PBindRelation (Figure 5) that lets users associate an identifier with a persistent relation. Bind, unbind, and fetch operate on table, internally represented as a symbol table. Pers.retrieve retrieves the table, if it exists, that is bound to the identifier RELATION_TABLE; if
functor PBind_Relation ( Relation : RELATION ) : BIND_RELATION = struct

structure Relation : RELATION = Relation
  type identifier = Table.identifier
  exception UnboundId = Table.UnboundId

  fun new_table () =
    let val st = Table.new ()
    in
      (Pers.bind ("RELATION_TABLE", st);
       Pers.save ();
       st)
    end

  val table = (Pers.retrieve "RELATION_TABLE") : Table.symtable
    handle Pers.UnboundId => new_table ()

  fun bind (rel, ident) = (Table.bind table (ident, rel);
                            Pers.save ()

  fun unbind ident = (Table.unbind table ident;
                        Pers.save ()

  fun fetch ident = Table.retrieve table ident
end

Figure 5: Bindable Persistent Relations

it does not exist, then through the call to new_table we create a new table, bind it to RELATION_TABLE, save it, and return it.

To store, remove, and retrieve bindable persistent relations, users make calls on the externally visible bind, unbind, and fetch functions. Bind lets users associate an identifier with a relation. It adds this binding to the internally named table, RELATION_TABLE. Unbind lets users break the binding between an identifier and a relation and fetch lets users retrieve a relation associated with an identifier.

By applying these two functors to the previously created relation structure, Relation (Section 1.3), we can now create a persistent relation structure, PRelation, and a bindable persistent relation structure, PBind_Relation:

structure PRelation = PRelation(Relation);
structure PBind_Relation = PBind_Relation(PRelation);

If we create, using PRelation.create, a persistent relation, pr, our implementation guarantees that changes resulting from subsequent PRelation.inserts and PRelation.deletes to pr are persistent. We achieve orthogonality between type and persistence: pr is of type relation to which we can perform the same operations as for any relation. Similar remarks hold for any bindable persistent relation created using PBind_Relation.create.
signature UNDO = sig
exception Restore of exn
  val checkpoint : (unit -> 'a) -> 'a
  val restore : exn -> 'a
end

Figure 6: Signature for Undo

To show how we use the persistent environment, suppose we create a bindable persistent relation, bpr, and then add it to the persistent environment:

PBind_Relation.bind bpr "MyPersistentRelation";

Then we can quit this SML session and later retrieve the saved relation into bprl using:

val bprl = PBind_Relation.fetch "MyPersistentRelation";

The simplicity of our approach raises a namespace problem with identifiers used in the persistent environment itself (i.e., the persistent symbol table mapping identifiers to persistent values). For now, we assume that for each type T, we can use the identifier T.TABLE to keep track of all persistent values of type T. Of course, as illustrated above with our examples using pr and bpr, programmers who simply want to create and make persistent values of type T never see or need to know about the name T.TABLE.

3. Undoability

Undoability means that any change to a program’s store can potentially be undone. This property is only of relevance in the presence of side-effects. Support for undoability requires the ability to save a program’s store and restore a program’s store to a previously saved one.

3.1. Model of Computation

Informally, a program’s store is a mapping between locations and values. Formally, SML defines the semantic domain Mem to be the set of finite mappings from Addr (memory locations) to Val; a store is an element of Mem. As an SML computation proceeds, most changes are to the environment, not the store, since SML programs are mostly functional. However, through assignment to ref values, users can make explicit changes to a program’s store.

3.2. Interface

The UNDO signature shown in Figure 6 provides an interface for users to undo changes to the store.

It provides two operations that checkpoint and restore the store. In the normal case (non-exceptional), checkpoint has the identical effects of simply calling its functional argument $f$; that is, all changes to the current store by $f$ are in effect upon return, and if executing $f$ returns a value or raises an exception so does executing checkpoint $f$.

The call restore $e$ has the effect of resetting the store to the (dynamically) previously checkpointed store and raising

9
the exception Restore with value $e$. A call to \textit{restore} always returns control to the point at which the store was last checkpointed; we effect this flow of control using SML's exceptional handling mechanism.

Because of this transfer of control by \textit{restore}, \textit{checkpoint} can also terminate by raising the Restore exception. Hence, when the Restore exception is raised as a result of a call to \textit{checkpoint}, it is as if no change to the current store has been made. This functionality of \textit{checkpoint/restore} will give us the ability to support the "all-or-nothing" property of transactions.

The rationale for providing an exception Restore is to distinguish between a normal return (from \textit{checkpoint}) where side effects are done and one in which \textit{restore} is called, in which case side effects are undone. Having the Restore exception return an exception value is useful since it lets \textit{restore}'s caller pass information through the \textit{restore} back to the caller of \textit{checkpoint}. Moreover, as we will see in detail in the next section, it provides us with a nice way to handle transactional semantics.

By means of foreshadowing, as a simple example, consider the following function\(^5\):

\[
\text{fun foo () =}
\]

\[
(x := 5;
\quad \text{if C then Undo.restore Abort}
\quad \text{else !x})
\]

where $x$ has been defined and Abort is an exception value (in anticipation of the next section). In the following call to \textit{foo}, let $st$ and $st'$ be the values of the store before and after the call:

\[
(\text{Undo.checkpoint foo}) \text{ handle Restore exn => [some work]}
\]

When we call \textit{foo} the current store is $st$. If C is false, the store is updated by the change to $x$, 5 is returned, and computation proceeds as usual with the updated store $st'$. If C is true then $st$ is unchanged, i.e., $st' = st$, the Abort exception is passed back, and \{some work\} is done (e.g., abort-handling code or reraising Abort).

In an earlier design of UNDO we explicitly exported the type \textit{store} and had \textit{checkpoint} and \textit{restore} take the store as an explicit argument. However, there really is only one store (the mapping from \textit{Addr} to \textit{Val}) and we cannot create or assign stores; in this sense, stores were not fully first-class [JD88]. Moreover, we placed restrictions on the usage of \textit{checkpoint/restore}; for example, we assumed the \textit{restore} was always called within the dynamic scope of a call to \textit{checkpoint}. Our more straightforward design now embodies this disciplined use of the store and disallows surprising behavior like jumping arbitrarily to any arbitrary store.

### 3.3. Implementation

To implement undo, we need to keep a log of all modifications to the store and the old values (elements of \textit{Val}) originally assigned to the modified locations (elements of \textit{Addr}). To restore the previous state of the store, we simply replay the log from youngest entry to oldest. To handle nesting, we need to remember intermediate points in the log; for single-threaded applications, we can follow a simple stack discipline to remember these points.

For traditional imperative languages with explicit storage management, this log-based approach has several

\(^5\) in SML is the \textit{fetch} operation on ref's.
drawbacks. First since modifications to the store would be frequent, maintaining and replaying such logs would be expensive. Second, since storage is managed explicitly, the undo system would have to maintain carefully copies of objects referred to by the undo log. This would be a formidable task, especially in languages where pointers and integers cannot be distinguished.

For SML and other mostly functional languages, using an undo log to implement undoability is much more reasonable. First, assignments are rare, and in fact happen to only a few data types, i.e., refs and arrays. Maintaining a log and replaying it is not prohibitively expensive. Second, since the garbage collector does storage management, it is easy to ensure that data referred to by the undo log are not deleted; we need only make sure that the garbage collector is able to reach the entries in the log.

3.3.1. Runtime Data Structures and Routines

The three main pieces of state information we maintain for our implementation of undo are the extended store list, a checkpoint stack, and the undo log. The four main activities in our implementation of undo for SML/NJ are log construction, checkpoint creation and deletion, garbage collector interaction with the undo log, and finally, log replay.

Log Entries, Log Construction

SML/NJ already logs the locations of mutations to pointers and arrays in a store list as part of its generational garbage collection strategy. As for our implementation for persistence, to implement undo logs, we extend this mechanism by modifying the code generated for mutations. We make two changes in creating our extended store list: First, rather than log only mutations that might affect the pointer graph (which the garbage collector uses) we also log entries for mutations to non-pointer values, i.e., integers and byte arrays. Second, rather than log only the location of these mutations, we must also record the old values. For tagged data types (integers and pointers), storing these values is easy: We just add an additional old value field to the record as used in the original store list; the old value is tagged, and thus is acceptable to the garbage collector. For (mutable) byte arrays, it is more difficult since the old value is a full 32-bit quantity, and thus is "untagged"; in this case, we allocate a new record to hold the old value. We use a tag for this record to inform the garbage collector not to look at the 32-bit quantity it contains. We call these extended records undo log records since their old value fields will be used for undoing the store. Finally, entries are prepended to the extended store list, thus ordering them from new to old.

The undo log (which we represent as a list) is initially empty. As we discuss in detail below, a side effect of calling restore is to do a garbage collection and a side effect of doing garbage collection is to prepend the extended store list to the current undo log. In this way, the undo log grows, and in the correct order (newest to oldest).

Checkpointing

Since we support nested checkpoints and restores, we maintain a stack of checkpoints, each of which points to an undo log record (either on the extended store list or on the undo log). When we establish a new checkpoint through a call to checkpoint, we call a new runtime function that pushes a new pointer on the stack of checkpoints; this checkpoint
points to the most recent entry in the extended store list. After a nested checkpoint terminates, we pop this stack. After the last checkpoint terminates, we discard the entire undo log. Since we maintain the undo log in the garbage collected heap, we do not have to deallocate the log explicitly; rather it is deallocated during the next garbage collection.

**Interaction with the Garbage Collector**

The trickiest aspect of the undo system involves the transfer of the store list to the undo system during garbage collection, and the subsequent garbage collection traversal of the entries in the undo log. Recall the generational garbage collector uses the store list (and hence, our extended store list) and normally discards it after each minor collection. In our support for undo, at each garbage collection we start the normal generational garbage collection traversal of the extended store list by examining only the items which would have appeared on the original store list. Then we complete the normal garbage collection step by computing the transitive closure of the pointer graph. At this point we have examined exactly the same storage locations as in a standard garbage collection (without our extended store list). Now, instead of discarding the store list, we pass it to the undo system, which prepends it to the undo log.

Finally we start another garbage collection using as roots the appropriate pointers in the undo log's entries. Since the undo log is just a normal list to the garbage collector, it gets copied, as well as any data to which it refers that have not been previously copied. This garbage collection step preserves all of the old values, even if they are no longer referred to by other data structures in the heap. This two-phase approach allows us to measure the storage overhead imposed by the undo system by simply looking at how much data is copied in the second phase.

**Replay**

Before replaying the log, we first force a garbage collection to occur. As just explained, this has the side effect of prepending more entries onto the undo log. Next we replay the log from youngest to oldest, rewriting old values, until we find the checkpoint that matches the top of the checkpoint stack. Finally we pop the checkpoint stack.

### 3.3.2. Costs

Section 5 discusses specific benchmark results that suggest that the costs of maintaining an extended store list and an undo log are not overwhelming.

Since most data in SML/NJ are each represented as a pointer, we need only copy one pointer per data object; we also do not inappropriately garbage collect values saved on the undo list. Hence, to a first approximation the only costs we incur in maintaining an extended store list are in doing additional pointer copies and the inability to garbage collect old values.

The cost in doing a restore involves replaying the log. This may sound expensive, but mutations in SML programs are rare: the lists are typically short. Also, restoring an old value only involves restoring the pointer, since the old value will not have been garbage collected. Again, we let regular garbage collection clean up values stored on the undo log.
functor URelation (Relation : RELATION) : RELATION = struct

  fun insert (tup, rel) =
    let
      fun restorer () =
        Relation.insert (tup, rel) handle exn => Undo.restore exn
      in
        (Undo.checkpoint restorer) handle Undo.Restore exn => raise exn
      end
    end

  fun delete (tup, rel) =
    let
      fun restorer () =
        Relation.delete (tup, rel) handle exn => Undo.restore exn
      in
        (Undo.checkpoint restorer) handle Undo.Restore exn => raise exn
      end
    end

end

Figure 7: Undoable Relations

3.4. Use: Undoable Relations

Figure 7 shows the interface for "undoable" relations 6, by extending the signature for relations. Again, the two relevant operations are insert and delete. We wrap the call to Relation.insert by a checkpoint of the store before the call using checkpoint and a handler for the Restore exception, in case an exception is raised. If executing Relation.insert raises any exception e (e.g., InvalidRTuple), we call Undo.restore, which causes the Restore exception with e as its exception value to be raised and control to transfer to the point at which checkpoint was invoked; the outer handler catches the Restore exception and rethrows e. The code for delete is similar.

We can create an undoable relation structure by applying the functor to our Relation structure from before:

structure URelation = URelation(Relation);

If we create an undoable relation, ur, using URelation.create, then if an exception is raised from attempting to insert into or delete from ur, the effects of the insertion or deletion are undone.

4. Transactions

As mentioned in the introduction, a transaction is a group of operations that is treated atomically ("all-or-nothing"). That is, a transaction must be atomic and permanent. Atomicity means that a transaction either succeeds completely and commits, or aborts and has no effect. Permanence means that the effects of a committed transaction survive failures. In the presence of concurrency, transactions must additionally be serializable, which means that concurrent transactions must appear to execute in some serial order. With nested transactions, a transaction's effects become permanent only when commit occurs at the top-level. That is, the permanence of effects of a nested transaction is

---

6For completeness, the Appendix contains the module for bindable undoable relations. They do not serve any purpose in the discussion here, but emphasize the orthogonality between persistence and undoability.
signature TRANSACT = sig

  exception Abort

  val transact: (unit -> 'a) -> 'a

  val abort: unit -> 'a
  val abort_top_level: unit -> 'a

end

Figure 8: Signature for Transact

relative to its parent's commit.

By putting the support for persistence and undoability together, we can provide support for single-threaded nested transactions. Support for persistence gives us a way to guarantee permanence and support for undoability gives us a way to guarantee atomicity. We are deliberately not handling concurrency in this paper, and thus, can ignore serializability.

4.1. Model of Computation

We combine the additions to the model of computation for persistence and undoability. We extend SML state to include the persistent memory, $PMem$, and we extend the SML environment to include the persistent environment, $PEnv$:

$$
State = Mem \times \ldots \times PMem
$$

$$
Env = \ldots \times PEnv
$$

4.2. Interface

Figure 8 gives the TRANSACT signature. The function transact called with a function $f$ has the effect of executing $f$ atomically. It begins a possibly nested transaction, which commits if and only if $f$ returns without raising an exception; we treat exceptional termination of a transaction as an abort. If the committing transaction is top-level, all its changes to the persistent environment and persistent memory are saved to disk. If the committing transaction is not top-level, no changes to the persistent environment or persistent memory are made. If a transaction aborts, all (and, in the case that it is nested, only) its changes are undone. These properties of transact ensure that the permanence of the effects of a child transaction depends on the commit/abort of its parent. Only at the top-level do effects of committed transactions get saved to permanent storage, i.e., written to disk.

A call to abort has the effect of raising the Abort exception and undoing a transaction's effects by one level. A call to abort_top_level has the effect of raising the Abort exception at the top-level and undoing the effects of the top-level transaction, including the effects of all its nested transactions. As with any exception, transact's caller can use an explicit handler for the Abort exception, e.g., if the abort of a nested transaction is not to propagate.
Consider a simple example:

```sml
fun foo () =
  ( x := x + 1;
    if C then !x else raise Abort )

fun bar () =
  ( x := x + 2;
    if D then ( Transact.transact foo; !x ) else raise Abort )
```

and the following calls to `transact`:

```sml
Transact.transact foo;
Transact.transact bar;
```

In the first call, if C is true then x is incremented and the new value is returned; otherwise, x remains unchanged and the Abort exception is raised. To show how nesting works, consider the second call: If D is true then if C is true, x gets incremented by 3; if D is true and C is false, x gets incremented by 2; if D is false, x remains unchanged and the Abort exception is raised.

### 4.3. Implementation

The implementation of the TRANSACT signature is entirely in SML using the interfaces provided by PERS and UNDO. Figure 9 gives the code.

Conceptually, `transact` is the composition of two functions, g and f, where g has the main effect of checkpointing the current store (using `checkpoint`) and f has the effect of doing a nested transaction (do_trans) or top-level transaction (do_top_trans). In both the nested or top-level cases, if an exception is raised, then we call `restore` to undo the transaction's effects. In the case of a top-level transaction, we need to do a little more work: upon commit, we need to `save` all its changes to the persistent heap (i.e., persistent environment and persistent memory).

Let us now step through the code in more detail. First, we initialize a global boolean flag, in_transaction, that remembers whether or not we are inside a transaction already. Skipping down to the bottom of `transact`'s definition (at the line beginning do_check if ...), we test to see whether we are in a transaction; if so we return the function `do_trans`; otherwise, we return `do_top_trans`. We use `do_check` to checkpoint the current store and to handle the Restore exception, re-raising its exception value, `exn`, to `transact`'s caller.

Next, let us consider what `do_trans` does since for both top-level and nested transactions we eventually call it. `Do_trans` executes the closure argument to `transact`. If an exception `exn` is raised, the transaction aborts, restoring the store to the previously checkpointed value and raising the Restore exception with the exception value `exn`; control returns to the point at which the store was last checkpointed.

`Do_top_trans` first toggles the boolean flag and calls `do_trans` to execute the transaction's closure. `Do_top_trans` may complete successfully, thereby committing, or unsuccessfully, thereby aborting. If it commits (skipping the exception handler code), we save its effects in the persistent heap \(^7\), reset the boolean flag, and return the value, `res`, obtained as the result of executing the closure. If it aborts, it terminates with either an AbortTopLevel exception or by some other exception. If it terminates with an AbortTopLevel exception, then we restore the store and raise the Abort

\(^7\)The exception handled by the call to `save` is SaveFailed. See Section 2.
structure Transact: TRANSACT = struct
    exception Abort
    exception AbortTopLevel

val in_transaction = ref false

fun transact closure = let
    fun do_check f = (Undo.checkpoint f) handle Undo.Restore exn => raise exn
    fun do_trans () = closure () handle exn => Undo.restore exn
    fun do_top_trans () = let
        val _ = in_transaction := true
        val res = (do_check do_trans) handle AbortTopLevel => Undo.restore Abort
            | exn => Undo.restore exn
            in
            (Pers.save () handle exn => Undo.restore exn;
             in_transaction := false;
             res)
        end
    in
    do_check (if !in_transaction then do_trans else do_top_trans)
    end

fun abort () = raise Abort
fun abort_top_level () = raise AbortTopLevel
end

Figure 9: Implementation of Transact

exception. If it terminates with any other exception, we restore the store and reraise the exception. By restoring the store we treat any exceptional termination of a nested transaction as an abort, yet give the handler the opportunity to execute abort handling code depending on what kind of exception is raised.

Note that for top-level transactions there are two do.checks. The inner one allows us to convert an AbortTopLevel exception to Abort, to restore the store to the appropriate value, and to transfer control to the outermost do.check. The outer do.check will return control back to the caller of the top-level transaction. Without the innermost do.check, if an abort to the top level occurs, then because of the implicit transfer of control in restore, the call to restore in do.trans would bypass the handler for AbortTopLevel.

Our implementation handles the abort of a transaction to the top level (e.g., if some user code calls the abort_top_level function within a deeply nested transaction) by unrolling “inside-out” the effects of each nested transaction one level at a time, propagating the AbortTopLevel exception all the way until the outermost handler. Since we do not want or need to expose the AbortTopLevel exception we mask it by raising the Abort exception to the original caller of transact. We could have optimized the unrolling by handling the AbortTopLevel exception specially in the do.trans function, but it would make the code harder to read.

4.4. Use: Transactional Relations

Figures 10 and 11 show the interfaces for transactional and bindable transactional relations. The changes for insert,
functor TRelation (Relation : RELATION) : RELATION = struct
...

fun insert (tup, rel) =
  let fun wrapper () = Relation.insert (tup, rel)
  in
    Transact.transact wrapper
  end

fun delete (tup, rel) =
  let fun wrapper () = Relation.delete (tup, rel)
  in
    Transact.transact wrapper
  end
end

Figure 10: Transactional Relations

functor TBind_Relation (Relation : RELATION) : BIND_RELATION = struct
  structure Relation : RELATION = Relation

  type identifier = Table.identifier
  exception UnboundId = Table.UnboundId

  val table = (Pers.retrieve "RELATION_TABLE")::Table.symtable
    handle Pers.UnboundId => new_table ()

  fun bind (rel, ident) =
    let fun add () = Table.bind table (ident, rel);
    in
      Transact.transact add
    end

  fun unbind ident =
    let fun del () = Table.unbind table ident
    in
      Transact.transact del
    end

  fun fetch ident = Table.retrieve table ident
end

Figure 11: Bindable Transactional Relations
delete, are simple: we wrap the call to each corresponding Relation function inside a call to Transact.transact. The TBind_Relation functor is similar to the PBind_Relation functor (Figure 5), where for the creation of a new table and the bind and unbind functions we wrap calls to the (internal) Table functions within calls to Transact.transact. Each call to Transact.transact has the side effect of calling Pers.save with the new table so we do not need to make an explicit call to it here.

Again, through functor application, we can create two new structures:

```sml
structure TRelation = TRelation(Relation);
structure TBind_Relation = TBind_Relation(Relation);
```

Given a bindable transactional relation value, btr, we are guaranteed that a call to any TBind_Relation function like insert will be atomic. Moreover, if prog is a sequence of operations on btr and we call Transact.transact prog, then we are guaranteed that all of prog's effects will be done if this top-level transaction commits, or none are done if it aborts.

Finally, for completeness, we can add btr to the persistent environment:

```sml
TBind_Relation.bind btr "MyTransactionalRelation";
```

We can quit this SML session, and in a later session retrieve the saved relation into btrl using:

```sml
val btrl = TBind_Relation.fetch "MyTransactionalRelation";
```

5. Current Status and Performance Results

5.1. Current Implementation Status

All the code given in this paper runs. In short, persistence with RVM works, undoability works, and nested transactions work. The implementation includes approximately 200 lines of new SML code and modifications to about 80 lines of existing SML code; 850 lines of new C code and modifications to 250 lines of existing C code.

We plan to hide init's filename arguments as "command-line" arguments so users can link to one of many persistent heaps at start-up. We have not yet implemented the stop-and-copy garbage collection of the persistent heap.

5.2. Preliminary Benchmark Results

To determine what overhead our persistence and undo facilities add to SML/NJ, we have run preliminary benchmarks on two examples: the relation example as presented in this paper and the SML/NJ compiler itself. Our results indicate that we can perform 1-2 transactions per second which is acceptable performance for our application domain. Most of the cost in persistence is time spent on scanning the persistent heap; most of the cost in undoability is in garbage collection—doing collection more frequently and copying additional data values. These results suggest places in our implementation that warrant optimizations for future work. Since we have not done extensive testing, the following analysis of our results is not definitive, merely suggestive.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>10 ruples</th>
<th>100 ruples</th>
<th>1000 ruples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. plain_orig</td>
<td>13.3</td>
<td>133</td>
<td>4500</td>
</tr>
<tr>
<td>2. plain</td>
<td>19.9</td>
<td>214</td>
<td>5230</td>
</tr>
<tr>
<td>3. undo_coarse</td>
<td>22.1</td>
<td>429</td>
<td>41000</td>
</tr>
<tr>
<td>4. undo_fine</td>
<td>21.8</td>
<td>418</td>
<td>40900</td>
</tr>
<tr>
<td>5. undo_fine_only</td>
<td>21.9</td>
<td>223</td>
<td>5380</td>
</tr>
<tr>
<td>6. undo_coarse_fail</td>
<td>71.3</td>
<td>887</td>
<td>45600</td>
</tr>
<tr>
<td>7. undo_fine_fail</td>
<td>71.5</td>
<td>892</td>
<td>45400</td>
</tr>
<tr>
<td>8. undo_fine_only_fail</td>
<td>22.6</td>
<td>225</td>
<td>5320</td>
</tr>
<tr>
<td>9. pers</td>
<td>306</td>
<td>574</td>
<td>6000</td>
</tr>
<tr>
<td>10. trans_coarse</td>
<td>427</td>
<td>1330</td>
<td>49300</td>
</tr>
<tr>
<td>11. trans_fine</td>
<td>421</td>
<td>1090</td>
<td>39200</td>
</tr>
</tbody>
</table>

Table 1: Benchmarks for Relation Example (times in milliseconds)

The Relation Example

For the relation example, we ran eleven cases (see Table 1). The first two cases give baseline measurements. In the remaining cases, for each, we create a bindable relation, \( R \), insert \( n \) rtypes, for \( n = 10, 100, \) and \( 1000 \), into \( R \); and bind \( R \) to an identifier; the undo cases differ in when we do checkpoints, how many we do, and whether we do any restores. Case 9 measures persistence only; cases 11-12 measure both persistence and undo as invoked through `transact`.

We now look at these cases in detail. All times are in milliseconds. The first case shows how much time in the original runtime (SML/NJ version .67) it takes to do the creation, inserts, and binding. The second case shows how much overhead is added by having to maintain the extended store list and by having our changes for persistence and undo compiled in, but not exercised. The time goes into creating the extended store list and traversing it when garbage collection occurs. This second case should be used as a basis for comparison for the remaining cases.

In the `undo_coarse` case, we take one global checkpoint before the creation, insertions, and binding; we do no restores. In `undo_fine` we additionally take a checkpoint before the insertion of each rtuple and the final binding; again, we do no restores. Comparing both cases to `plain`, we see that we incur little overhead when small amounts of data are being recorded (on the extended store list). We suspect the overhead is due to garbage collection, and for two reasons: collections are performed more frequently and more data is copied during each collection. As the number of rtypes increases, more data is being copying. It is not surprising that there is little difference between cases 3 and 4 since the same amount of data has to be stored on the extended store list, traversed, and copied; case 4 essentially shows that doing nested checkpoints incurs little overhead. In `undo_fine_only` we do no initial global checkpoint, but rather take checkpoints only before each insertion and the final binding. In this case we see a difference only as the amount of data checkpointed increases. As the amount of data increases, we need to do more garbage collections and more

---

8 Of the persistent, undoable, or transactional variety, as the case may be.
copying. Since the amount of data being copied does not change across cases 3-5, case 5 indicates how much time is spent in doing additional collections.

The next three test cases (6-8) are similar to the previous three (3-5), except restores are done. Comparing these "failure" cases with their "successful" counterparts, we believe the additional overhead for doing a restore is due to a fixed amount of time it takes to do a minor garbage collection plus the time it takes to actually do the restore (traversing the undo log and copying old values). There is roughly a linear relation in the number of tuples (4.5 ms./tuple), given that a minor garbage collection takes about 5 ms. (which becomes insignificant as the number of inserted tuples increases).

The pers case indicates the baseline cost for doing a save after the binding of the relation. Time is spent both on scanning the entire persistent heap and writing to disk.

Finally, the two transact cases (taken together) indicate a roughly additive relation between undo and persistence as intuition would lead us to believe: the cost of doing a transaction is approximately the same as the sum of the cost to do a checkpoint, the cost to do a save, and some fixed amount of overhead.

The SML/NJ Compiler

We ran a second set of benchmarks on the SML/NJ compiler itself. Recompiling the compiler, which is about 45,000 lines of SML code, with none of our features compiled in takes 2081.6 seconds of elapsed time. With our features included, it takes 2188.2 seconds; hence maintaining the extended store list adds only about a 5% overhead in time.

The compiler is more representative of a typical SML program, one that is mostly functional, compared to our highly imperative relation example; the compiler is also a large, "useful" program. Thus, our preliminary benchmarks suggest that for SML applications that use transactions, support for undoability does not incur a large performance penalty and that the main expense is the price paid for saving the persistent store.

6. Related and Future Work

6.1. Related Work

What primarily distinguishes our work from others is the principle of orthogonality between the persistence and undoability properties of transactions. No other language pulls out so explicitly the undoability property from transactions as we do; rather, more typically, "save" implies transaction commit and "undo" implies transaction abort. By our separating the two properties, we can distinguish between persistent memory (PMem) from regular memory (Mem) (this is what persistence provides) and between doing and undoing effects to memory, persistent or otherwise (this is what undoability provides). Then we can put the two properties together to give transactional semantics.

Support for just persistence needs no motivation as witnessed by the existence of a multitude of persistent programming languages and systems (e.g., PS-Algol [ABC*83], Napier [MBC*88], Poly/ML [Mat87], Amber[Car86], Galileo [ACO85], Exodus [CDG*90], Argus [LS83], Avalon [DHW88], Mneme [MS88]; see [AB87] for a survey).

Support for undoability, aside from transactions, is useful for applications like interactive debuggers [TA90],
backtracking programs, and database systems using optimistic concurrency control; they share a need to save the state of the system, e.g., by checkpointing a store, and to go forward and backward in time. Johnson and Duggan give a denotational semantics for first-class stores; they use a version stamp scheme on persistent data structures to implement stores as first-class objects [JD88]. Wilson and Moher propose a general callcs (call-with-captured-state) construct, similar to Scheme's and SML/NJ's call/cc (call-with-current-continuation) construct, that lets one treat stores as first class [WM89]. Their callcs construct, use of garbage collection techniques for implementing checkpointing, and other ideas about demonic memory inspired our initial design of UNDO's interface. As noted in Section 3, we backed off from callcs's full generality for ease in understanding and implementation, and most importantly, to enforce greater safety.

Our work relates most closely to two classes of programming languages: persistent languages and transaction-based languages. Some persistent programming languages, such as Poly/ML [Mat87] and Amber [Car86], do not support transactions at all, except perhaps implicitly as the top-level interactive session with a user. Others, such as PS-Algol [ABC*83] and Napier [MBC*88], support a simple database-oriented notion of a transaction where the act of opening a database file for writing begins a transaction and the act of closing it commits it. Explicit routines for committing and aborting may also be available, but users have little other control over transaction management. A transaction’s role as a control abstraction is combined with its role as part of the database file abstraction. We choose to treat transactions only as a control abstraction.

General-purpose transaction-based languages like Argus [LS83] and Avalon [DHW88, CLNW90] do not decouple the persistence and undoability properties of transactions. Atomic data types give users a means of guaranteeing both properties and they are inseparable. Argus and Avalon also do not support the principle of orthogonal persistence; e.g., array and atomic array are both built-in types in Argus. However, both Argus and Avalon handle concurrency and guarantee strong correctness conditions, e.g., dynamic atomicity (Argus) or hybrid atomicity (Avalon) [Wei89], to clients of atomic data types. Our mechanisms lie at one level lower since others are now free to extend our work, providing whatever concurrency correctness condition they desire.

Of the database-oriented programming languages (e.g., Pascal/R [Sch83], Adaplex [SFL83] and Taxis [MBW80]), because of its type system and base language (ML), the most closely related is Galileo [ACOS85]. Its idea of extending the global environment with additional bindings through the use construct is similar to our use of SML’s module facility, in particular functor application, to extend SML’s top-level environment; e.g., in the case of persistence, we add and remove bindings to and from the persistent environment, which is just an extension of the top-level SML environment. Galileo does not explicitly provide an “undo”-only facility, but it does have limited support for transactions. It supports top-level transactions implicitly (every top-level expression is executed atomically) and nested transactions explicitly through the transaction/end transaction bracketing construct. The only way to abort a transaction is to raise an exception. Because it is database-oriented, atomicity is guaranteed against the database, which serves as the storage mechanism for persistent data, rather than smaller chunks of data; however, programmers can use its class and subclass

---

9 Avalon provides a class called recoverable which is similar to providing just persistence but programmers are encouraged to use it only in constructing atomic data types.
features to gain finer-grained control over data.

In the introduction we motivated our work by identifying the problem of a mismatch between an application's needs and a transaction's guarantees. An alternative approach to addressing this mismatch is to relax the guarantees that transactions provide. Different correctness properties such as relaxed atomicity [LKS91] do not guarantee strict serializability; applications may be willing to tolerate inconsistent states temporarily for the sake of availability [S*90]. Here, we have assumed that our applications need full transactional semantics. Clearly, this related work complements ours and it would be interesting to see how we could modularize and then combine these properties to obtain weaker notions of correctness.

6.2. Ongoing and Future Work

Support for Heavyweight and Lightweight Concurrency

We have already built, but not yet thoroughly tested, mechanism to support concurrent transactions (multiple "heavyweight" processes). We use standard two-phase read/write locks to ensure serializability among concurrent transactions. The implementation essentially keeps locking information per transaction state. We support Moss's rules for nested concurrent transactions [Mos81].

Along with others at Carnegie Mellon, we have separately designed and built a Threads package for SML/NJ [CM90]. We have begun to integrate this Threads package with our support for persistence, undoability, and transactions. For example, we can run multiple threads of control, each of which does multiply nested checkpoints and restores. This demonstrates the orthogonality between lightweight concurrency and undoability.

Support for Distribution

We would like to build support for distributed transactions. Currently at CMU only rudimentary support for distributed SML in Mach [A*86] exists; essentially an SML interface exists for each Mach inter-process communication interface as written in C. We would like to provide a much more abstract view of distributed computation for SML where any SML value, including large complex data structures, user-defined abstract values, functions, and closures would be transmissible. Again, to a first approximation, we view distribution as a feature orthogonal to persistence and concurrency, and hope to provide support for distribution through module composition, modifying the runtime as necessary where the features inherently interact.

SML-specific Work

Currently the SML community is interested in adding dynamic types to SML, which we could exploit not only for persistence (see Section 2), but also for distribution. There is also interest in making SML modules "first-class." Adding this feature would let us store structures, i.e., environments, in persistent memory. It would also enable us to treat persistent environments uniformly as any other environment. As mentioned, we can view a persistent environment as simply an extension of SML's top-level environment.

We have yet to give our extensions to SML a completely formal static and dynamic semantics, though we intend to
build upon existing literature [MTH90, DHM91, ACPP91]. The semantics for persistence would be straightforward; however, that for undoability would be more challenging because of the need to track additional "potential" changes to state and control flow. As implied by the point of this paper, we can then give a semantics for \texttt{transact} in terms of that for persistence and undoability.

7. Acknowledgments

For fruitful discussions of our design, we thank Anurag Acharya, Eric Cooper, Bob Harper, Peter Lee, Greg Morrisett, Benjamin Pierce, Gene Rollins, and David Tarditi. For helpful comments on an earlier draft of this paper we thank other members of the Venari Project: Manuel Faehndrich, Greg Morrisett, Gene Rollins, and Amy Moormann Zaremski. We owe a special thanks to Manuel Faehndrich for his invaluable contributions and participation in implementing extensions to SML/NJ during the Summer of 1991.
8. Appendix

Below is the functor defining bindable undoable relations.

functor UBind_Relation(Bind_Relation : BIND_RELATION):BIND_RELATION = struct
    structure Relation : RELATION = Bind_Relation.Relation
    type identifier = Bind_Relation.identifier

    exception UnboundId = Bind_Relation.UnboundId

    fun bind (rel, ident) =
        let fun restorer () = Bind_Relation.bind (rel, ident)
            handle exn => Undo.restore exn
        in
            Undo.checkpoint restorer
        end

    fun unbind ident =
        let fun restorer () = Bind_Relation.unbind ident
            handle exn => Undo.restore exn
        in
            Undo.checkpoint restorer
        end

    end

    val fetch = Bind_Relation.fetch

end
References


26


