Appears in *PhysComp96*
Workshop on Physics and Computation

A logically reversible evaluator for the call-by-name lambda calculus

Lorenz Huelsbergen
lorenz@research.bell-labs.com
Bell Laboratories, Lucent Technologies

We present an evaluator for the call-by-name lambda calculus that is *logically reversible*. This evaluator is therefore a candidate for a computation mechanism that does not dissipate energy due to information erasure. Our design extends Landin's SECD evaluator for the lambda calculus with a history tape H—as in Bennett's constructions of reversible Turing machines—and is hence called SECD-H. The history tape records sufficient information to supply each state of a computation with a unique predecessor state. H can therefore be used to reverse the computation and to reset the machine. Since the lambda calculus is a model for realistic programming languages, SECD-H can, unlike a reversible TM, be directly used to simulate reversible computation of large and complex programs. We provide a complete implementation of SECD-H for such purposes.

1 Introduction

Logically reversible machines are candidates for low-power computation because such machines never erase information and hence can compute while dissipating only arbitrarily small amounts of energy[3, 7]. Every state in a reversible[1] computation must have—in addition to a unique successor state—a unique predecessor state. Deterministic computations on conventional computers have, for each of the computation's states, a unique successor state; however, they do not satisfy the reversibility requirement that every state have a unique predecessor state. The program,

```
if (x < 0)
    x = 0;
else
    x = x - 1;
```

running on a conventional non-reversible computer, serves to illustrate. If this computation terminates with `x` having the value 0, one cannot know whether the predecessor to the terminal state was the statement "`x = x - 1;`" (with `x` having initial value 1) or the statement "`x = 0;`" (with `x` initially being negative). This information has been irreversibly erased and dissipated as heat when `x`'s value changed. Programming models for describing reversible computations—significantly different from conventional models (such as the random-access-machine used above)—are needed.

Bennett[3] created a reversible Turing machine (RTM) that satisfies the above reversibility criteria of unique successor and predecessor states. By adding a history tape to a conventional Turing machine (TM), Bennett arranged for state transitions to save explicitly, on this history tape, any information that would otherwise be completely erased. After computing a result in this manner, the history tape is used to reverse the RTM and to "undo" the computation. The key point is that this history tape is initially blank (all bits initialized to 0) and that the operation of writing information on a blank tape is logically reversible. For example, if a register is written to a blank tape, the register can later be restored to this value (by reading it from the tape); the tape is restored by writing zeros to it—no information has been erased. This and other techniques for reversibly manipulating data can be found via Bennett's overview[4].

Turing machines—conventional and reversible alike—are powerful theoretical models. As such, however, they are too abstract for exploring reversible software in the large. It is difficult to provide a mapping from a practical program (such as a language evaluator or compiler, for example) to a TM. Furthermore, a TM's linear nature would make program simulation quite inefficient. One would however like to investigate reversibility at the scale of extant software systems. To this end, we describe a high-level language evaluator that meets the criteria of logical reversibility.

Our evaluator is based on the lambda calculus, a calculus of functions (described briefly in §2.1). It is well known that the lambda calculus is Turing complete. Furthermore, it forms the foundation for the family of *functional* programming languages (e.g., Lisp[9], ML[11, 10], Haskell[6]). In practice, functional languages are often compiled to machine language via the lambda calculus (e.g, [1]). In this paper we consider normal-order (call-by-name) reduction of lambda terms (*i.e.* programs). This reduction order stipulates that the arguments to a function are not evaluated when the function is applied (called), but rather only when the value of an argument is actually required (as a predicate in a conditional, for example). This is in contrast to applicative-order (call-by-value) re-

[1] For the rest of the paper, we will use the term *reversible* to denote *logical reversibility*.

duction where a function's arguments are evaluated before function application. Note that applicative-order evaluation corresponds to most current programming-language practice. We chose normal-order evaluation because the specification of the reversible machine is (slightly) simpler; a similar construction has also been devised for applicative-order reduction, but it is not described in this paper.

The normal-order lambda calculus serves as the programming language for our purpose—only a matter of syntax separates the lambda calculus from realistic functional programming languages. We now need a mechanism for evaluating lambda-calculus programs. Our goal is to render this mechanism reversible. Landin's SECD machine[8] is a non-reversible evaluator for the lambda calculus. We briefly describe SECD in §2.3. In a manner similar to Bennett's RTM construction[3], we extend SECD with a history tape H to yield a reversible lambda-calculus evaluator which we call SECD-H. We assume this machine to be finite in all respects; a computation that exceeds the resources of a given instance of SECD-H (e.g., out of memory) must be run on a larger, but still finite machine.

SECD-H computes similarly to Bennett's RTM. Given a terminating program as a lambda term e, SECD-H will evaluate e to produce a value v and a history tape h. After computing the output value v, this result is reversibly copied and retained as v'. Using e, v, and h, SECD-H then completely reverses the computation. This clears the history tape and thereby resets the machine. (The computation's result is still available as v'.) The reset SECD-H machine may now be used for further reversible computations. Note that the reset operation is crucial since the history is a finite resource (since the machine is finite). We discuss how SECD-H computations may trade time for space following Bennett's idea[3] of staged computations.

We use Standard ML (SML)[11, 10] to operationally specify evaluators in this paper. Since SML is an executable notation (e.g., via the SML/NJ ML compiler[1]), our construction directly gives an executable simulator for SECD-H with which experiments on reversible software can be conducted. This paper contains complete SML source for SECD-H (and SECD). We stress that this is only a simulator; low-power computation via SECD-H would require (at least) a reversible substrate (*i.e.*, hardware).

It is important to distinguish the various language components used in this paper and how they apply to our goal. We use SML to implement a reversible machine SECD-H derived from SECD. In turn, SECD-H reversibly evaluates the pure lambda calculus. Since functional languages are usually compiled to the lambda calculus—often only through a syntactic transformation—high level languages can be executed reversibly.

In the next section we briefly describe relevant lambda calculus, SML, and Landin's (non-reversible) SECD machine. Section 3 describes the reversible SECD-H machine proper. Section 4 discusses extensions to this machine to more expressive primitive types and how staged computations may trade space for time. We conclude with a summary and directions for future work in Section 5.

2 Language preliminaries

Here, we provide some language background (lambda calculus, SML, Landin's SECD) in brief. A comprehensive treatment of the lambda calculus is given by Barendregt[2]. SML is described in [11, 10] and in numerous text books (e.g., [12]). Field and Harrison[5] provide a contemporary description of Landin's SECD machine[8].

2.1 Lambda Calculus

The pure lambda calculus has a simple syntax for denoting a lambda term e:

e	::=	x	variables
	|	$\lambda x.e$	λ-abstraction
	|	$(e\ e)$	application

It consists of a set of identifiers denoting variables, λ-abstractions for defining functions, and applications of (function) terms to (argument) terms. The identity function, for example, can be written $\lambda x.x$ in the lambda calculus. As another example, the function *twice*

$$\lambda f.\lambda x.(f\ (f\ x))$$

takes two arguments, f and x, and applies f (necessarily a function) to another application of f to x (an arbitrary term).

It is well known that arbitrary data types—e.g., booleans, integers, tuples, lists—can be directly encoded in the lambda calculus. Control structures such as recursion and conditionals can be coded in the pure lambda calculus as well. Reference [5] provides a good explanation of such encoding schemes. To simplify construction of a reversible evaluator for the lambda calculus, we restrict ourselves to the pure lambda calculus—more expressive data and control structures can be synthesized from it. It is possible to extend the evaluators in this paper to support directly primitive types and operators on them (e.g., integers with addition, multiplication, comparison, etc.), but we do not do so here.

Normal-order evaluation in the pure lambda calculus proceeds in the context of an *environment* that maps identifiers to values. Values are *closures* formed by evaluating a term of the form $\lambda x.e$. A closure corresponding to $\lambda x.e$ is a triple that contains the current environment, identifier x and term e. In particular, a closure's environment

contains values for e's *free variables*; *i.e.*, variables not bound by a λ-abstraction in e. Evaluation of an identifier retrieves the value of the identifier in the current environment. Evaluation of an application $(e_0\ e_1)$ first evaluates e_0 which must result in a closure value v. A closure (also called a *suspension* in this context) s is then built for e_1; note that e_1 is not yet evaluated. The closure v is then applied by binding the suspension s to v's variable x and evaluating v's term e in v's environment. Below (§2.3), we make this procedure precise by coding it as an evaluation mechanism.

2.2 Standard ML

SML is a strongly typed, higher-order, mostly functional programming language[11, 10]. Since SML has a well-defined semantics[11], it a suitable vehicle for algorithm specification. We will use it to implement SECD and SECD-H evaluators.

An ML program contains type definitions that declare data types and sets of functions to manipulate data described by the types. In particular, the recursive `datatype` construct introduces a discriminated union of values. *Constructors* serve to tag a `datatype`'s possible values. A constructor may carry zero or more values; a constructor itself is a constant. For example, the `datatype` named `term` in the program of Figure 1 defines the syntax of the pure lambda calculus described above. The `ID` constructor carries an identifier; we use integers to enumerate identifiers. `LAMBDA` carries a pair holding an identifier and recursively a value of type `term`. `APP` carries a pair of terms denoting the functional term and its argument. The nullary constructor `MARK` does not directly correspond to lambda-calculus syntax; rather, it is used internally in our SECD implementation to represent a certain kind of partially evaluated term.

Functions in ML are introduced with the keyword `fun` followed by the name of the function and *patterns* for its arguments. An "=" separates the argument patterns on the left-hand-side (lhs) from the executable expression on the right-hand-side (rhs). A function consists of a series of such pattern-expression pairs. The first pattern to match a function's arguments will bind any variables in the pattern to the corresponding pieces of the arguments; it then proceeds to evaluate the rhs with these bindings in place. Pattern matching is equivalent to a series of conditional expressions. As we shall see, a reversible evaluator will need to retain information about which pattern matched in order to have sufficient information to determine a state's unique predecessor.

Two primitive ML data types are central to our exposition: tuples and lists. Tuple and list notation may be used in patterns; we make extensive use of this capability to describe state transitions. Tuples provide general records. The notation for a tuple of n elements is

$$(\mathtt{e}_1,\ldots,\mathtt{e}_n)$$

where an $\mathtt{e}_i$ is an ML expression that computes tuple element i. Lists are constructed via

$$\mathtt{e}_i::\mathtt{e}_j$$

where $\mathtt{e}_j$ must evaluate to a list l and $\mathtt{e}_i$ evaluates to an element to prepend (*cons*) onto l. ML denotes the empty list as `[]`. Lists are disassembled by matching them in patterns. We will use ML lists to represent various "tapes" in our evaluator.

It is important to note that when a tuple or list is created (and allocated) with recursive-data elements, it contains only *references* (*i.e.*, pointers) to this data rather than a complete copy of the data. Erasure of a tuple or list element erases only the references (and not the data structures proper). Since data structures are essentially chained references, reference erasure can incrementally remove data structures.

2.3 Non-reversible SECD Machine

Figure 1 contains an SML specification of Landin's SECD machine[8].[2] From this non-reversible evaluator for the pure lambda-calculus, we will derive a reversible evaluator in the next section. Here we describe major points of this SECD implementation.

Identifiers are represented as integers. Lambda-calculus terms are defined by the `term` datatype as previously described (§2.2). Values corresponding to evaluated lambda terms are implemented via the `value` datatype. Values are either closures or suspensions and carry a term and an environment; closures additionally carry a binding identifier. An environment is implemented as an association list of entries; each entry is an identifier-value pair. The empty environment is the empty list.

SECD consists of four pieces: a stack of values, an environment, a control list of lambda terms, and a dump (temporary storage for stack, environment, and control-list triples). As execution proceeds, successive terms from the control list are evaluated—their values placed on the value stack. The dump is used to save/restore the current configuration (stack, environment, control list) when evaluation enters a closure or suspension. The evaluation function `eval` operates on a 4-tuple representing the four components of SECD.[3] This function has seven rules corresponding to termination, restoration of saved state on the dump, lookup of an identifier's value, evaluation of a lambda-abstraction to a closure, application of a closure to an argument value, evaluation of an application term (entails suspension creation), and evaluation of a suspension. Except for the first rule, the rhs expressions call

[2] Our specification loosely follows that of Field and Harrison[5].

[3] The auxiliary function `eval'` conveniently provides an initial machine for evaluation (via `eval`) of its single argument term `t`; it is, however, not relevant to the subsequent discussion.

```
structure SECD =                                (* SECD machine *)
    struct
        type ident = int                        (* identifiers *)

        datatype term =                         (* lambda terms *)
            ID of ident
          | LAMBDA of (ident * term)
          | APP of (term * term)
          | MARK

        datatype value =                        (* values *)
            CLOS of (term * ident * env)
          | SUSP of (term * env)
        withtype entry = (ident * value)
        and env = entry list

        (* type abbreviations *)
        type stack = value list
        type control = term list
        type dump = (stack * env * control) list
        type secd = (stack * env * control * dump)

        (* environment lookup *)
        val emptyEnv : env = []

        fun lookup (i,((i',v)::e')) =
            if i = i' then v else lookup(i,e')

        (* term evaluation *)
        fun eval (v::[],[],[],[]) = v
          | eval (v::[],e,[],(s,e',c)::d) = eval (v::s,e',c,d)
          | eval (s,e,ID i :: c,d) = eval (lookup(i,e)::s,e,c,d)
          | eval (s,e,LAMBDA(i,t)::c,d) = eval (CLOS(t,i,e)::s,e,c,d)
          | eval (CLOS(t,i,e')::v::s,e,MARK::c,d) = eval ([],(i,v)::e',t::[],(s,e,c)::d)
          | eval (s,e,APP(f,v)::c,d) = eval(SUSP(v,e)::s,e,f::MARK::c,d)
          | eval (SUSP(t,e')::s,e,MARK::c,d) = eval ([],e',t::[],(s,e,MARK::c)::d)

        fun eval' t = eval ([],emptyEnv,t::[],[])

    end  (* structure SECD *)
```

Figure 1: SML specification of a (non-reversible) SECD machine.

```
val id    = LAMBDA(0,
                   ID 0)

val twice = LAMBDA(1,
                   LAMBDA(2,
                       APP(ID 1,APP(ID 1,ID 2))))

val e     = APP(APP(twice,id),ID 3)
```

Figure 2: Lambda terms using the `term` datatype.

`eval` recursively to evaluate "the rest of the program" after manipulating the SECD state.

The `lookup` function's argument is a pair consisting of an identifier and an environment (identifier-value list). It recursively traverses the environment from most recent binding to least recent until it finds a value for the argument identifier. It returns the identifier's value if it exists; otherwise, the term being evaluated is ill-formed and evaluation abnormally halts.

Sample lambda terms written using the `term` datatype are in Figure 2. The terms `id` and `twice` correspond to the pure lambda terms of Section 2.1. The term `e` applies the function `twice` to two arguments: `id` and identifier three (`ID 3`). Using an SML compiler and SECD from Figure 1, the terms of Figure 2 may be evaluated.

Examples of irreversibility abound in this machine. For example, the `lookup` function is not reversible since it discards its input environment—a value produced by `lookup` cannot be mapped to the input identifier. The rule for state restoration from the dump is also irreversible because the current environment `e` is discarded and replaced with `e'`. Similarly, the rule for identifier lookup discards information as to which identifier was retrieved. Although the rule for evaluation of a lambda-abstraction appears to be reversible (a `CLOS` closure value holds all of a `LAMBDA`'s components), this is not so because the data representing the actual `LAMBDA` on the control list will sometimes—when it is not part of some other term—become completely inaccessible by this rule. Similar irreversibility issues arise as well in the rules for closure application, application evaluation, and suspension application. The next section describes modifications (history tape, etc.) that render this machine logically reversible.

3 **Reversible** SECD-H

In this section we describe a reversible version of SECD. The key insight to our approach is that prepending an element onto a list with `::` is reversible. This is so because one can allocate list cells from a blank tape. Furthermore, it is straightforward to multiplex allocation for multiple lists from a single tape. In our scheme, "forward" computation will allocate data thusly; "backward" computation will reversibly remove head elements from lists (thereby clearing tape) at exactly the computation points at which they were initially written. Recall that elements containing recursive data do so with references; only references to recursive structures are erased in a deallocation step.

We extend SML syntax with the `:::` list constructor for patterns. When appearing in a pattern, this constructor signifies that reversible erasure of the element on the left is possible at this computation point and should occur. For example, the pattern $(x$ `:::` $l)$ indicates that x may be reversibly erased when this pattern is matched. Since `:::` is not SML syntax, simulation of the SECD-H evaluator with an SML compiler requires replacement of the `:::` constructor with `::`.

The implementation of SECD-H is contained in Figure 3 (type declarations) and Figure 4 (function definitions). We first describe SECD-H's type declarations and then its functions.

The `term`, `value`, `entry` and `env` datatypes of SECD-H are identical to the corresponding types in SECD. Furthermore, the reversible machine consists of the same stack, environment, control, and dump components as the irreversible machine. The first modification is the addition of the `event` datatype. This type contains constructors that discriminate between various actions of the machine; that is, between the selection of one evaluation rule over another. The events `ALPHA` and `OMEGA` respectively serve to demarcate the beginning and end of a computation. The other events carry information that would otherwise be lost but must be available for reversible erasure. Individual events are described below. A list of `events` constitutes the history and is embodied in a type named `history`. SECD-H is a quintuple—SECD plus `history`.

At this point we introduce some notation for reasoning about information containment. Let $x \subseteq y$ denote that y contains at least the information contained in x. (Note that if x' is a structural copy of x, then $x' \subseteq x \subseteq x'$). For example, where `e` and `e'` are SML identifiers denoting SECD-H environments, `e'` $\subseteq$ `e` means that data contained in `e'` is also completely contained in `e`.

Regarding functions, the first obvious difference is that for each function defined in the SECD evaluator, SECD-H provides two: a "forward" computing version and a "backward" computing version. Therefore, the SECD `lookup` function has been replaced with `lookup` and `lookupRev`; SECD `eval` has been replaced with `eval` and `evalRev` in SECD-H. Although a function in SECD-H may have the same name as one in SECD, its implementation is different from that in SECD. The SECD-H functions receive the history as a part of their argument and must correctly maintain it. We first consider the reversibility of the lookup operation and then that of the evaluation function.

```
structure SECDH =                              (* reversible SECD machine *)
    struct
        type ident = int                       (* identifiers *)

        datatype term =                        (* lambda terms *)
            ID of ident
          | LAMBDA of (ident * term)
          | APP of (term * term)
          | MARK

        datatype value =                       (* values *)
            CLOS of (term * ident * env)
          | SUSP of (term * env)
        withtype entry = (ident * value)
        and env = entry list

        (* type abbreviations *)
        type stack = value list
        type control = term list
        type dump = (stack * env * control) list

        datatype event =                       (* history *)
            ALPHA
          | OMEGA
          | ASSOC of env
          | RESTORE of (stack * env * dump)
          | LOOKUP of (env * control)
          | MKCLOS of control
          | APPCLOS of (stack * control)
          | MKSUSP of control
          | APPSUSP of stack
        type history = event list

        type secdh = (stack * env * control * dump * history)
```

Figure 3: Type declarations for the reversible SECD-H machine.

```
    (* environment lookup *)
    val emptyEnv : env = []

    fun lookup (i,e as ((i',v)::e'),h) =
        if i = i' then (e,h)
        else lookup (i,e',(ASSOC e)::h)

    (* reverse environment lookup *)
    fun lookupRev ((i,v)::e',h) =
        let fun aux (e',(ASSOC e)::h) = aux (e,h)
              | aux (e,h) = (i,e,h)
        in  aux (e',h)
        end

    (* term evaluation *)
    fun eval (s as v::[],[],[],[],h) = (s,[],[],[],OMEGA::h)
      | eval (s as v::[],e,[],d as (s',e',c)::d',h) = eval (v::s',e',c,d',(RESTORE (s,e,d))::h)
      | eval (s,e,c as (ID i)::c',d,h) =
        let val (e' as (i',v)::_,h) = lookup(i,e,h)
        in  eval (v::s,e,c',d,(LOOKUP (e',c))::h)
        end
      | eval (s,e,c as LAMBDA(i,t)::c',d,h) = eval (CLOS(t,i,e)::s,e,c',d,(MKCLOS c)::h)
      | eval (s as CLOS(t,i,e')::v::s',e,c as MARK::c',d,h) =
          eval ([],(i,v)::e',t::[],(s',e,c')::d,(APPCLOS (s,c))::h)
      | eval (s,e,c as APP(f,v)::c',d,h) = eval (SUSP(v,e)::s,e,f::MARK::c',d,(MKSUSP c)::h)
      | eval (s as SUSP(t,e')::s',e,c as MARK::c',d,h) = eval ([],e',t::[],(s',e,c)::d,(APPSUSP s)::h)

    fun eval' t = eval ([],emptyEnv,t::[],[],ALPHA::[])

    (* reverse term evaluation *)
    fun evalRev (s as v::[],[],[],[],OMEGA:::h) = evalRev (s,[],[],[],h)
      | evalRev (v:::s',e',c,d',(RESTORE (s,e,d)):::h) = evalRev (s,e,[],d,h)
      | evalRev (v:::s,e,c',d,(LOOKUP (e',c)):::h) =
        let val (i,e'',h') = lookupRev (e',h)
        in  evalRev (s,e,c,d,h')
        end
      | evalRev (CLOS(t,i,e'):::s,e,c',d,(MKCLOS c):::h) = evalRev (s,e,c,d,h)
      | evalRev ([],e',t:::[],(s',e,c'):::d,(APPCLOS (s,c)):::h) = evalRev (s,e,c,d,h)
      | evalRev (SUSP(v,e'):::s,e,f::MARK::c',d,(MKSUSP c):::h) = evalRev (s,e,c,d,h)
      | evalRev ([],e',t:::[],(s',e,c as MARK::c'):::d,(APPSUSP s):::h) = evalRev (s,e,c,d,h)

end  (* structure SECDH *)
```

Figure 4: Function definitions for the reversible SECD-H machine. The (non-SML) constructor ::: in a pattern signifies that the list element on its left may be reversibly erased.

In SECD-H, lookup receives as an argument a 3-tuple: an identifier i, an environment e in which to lookup i, and the current history h. If i is not at the head of the environment, lookup is called recursively with i and the tail e′ of the environment. Without a history, this recursive call could "lose" the reference to the head element of e, thereby making its reversible erasure impossible. By having the ASSOC constructor carry e, lookup guarantees that its return result contains the same information as its input argument. In particular, the environment e passed to lookup is contained in the history h returned from lookup ($e \subseteq h$) and, by the definition of lookup, $i \subseteq e \subseteq h$ where i is the identifier being looked up. Note that a computation that attempts to lookup an identifier not in the environment is not well-defined.

The function lookupRev reverses lookup's operation. As long as the history contains an ASSOC at its head, lookupRev rebuilds the initial environment from the environment e carried by the ASSOC event. The first pattern/expression clause in lookup's auxiliary function aux removes ASSOC events from the history and calls itself recursively to rebuild lookup's input environment. Note that in this clause aux can safely ignore and discard the first field e′ of its argument since lookup constructed e such that $e' \subseteq e$. The base case for aux returns the restored identifier, environment, and residual history.

We now consider the evaluation function eval and its inverse evalRev. Note that every pattern/expression clause in eval corresponds to exactly one history event and to one clause in evalRev. For any clause in evalRev, removal of the event from the history is reversible since the storage for the history is known to have initially been blank and we have a handle on a copy of any data carried by the event at this point in the computation. Use of the ::: constructor in patterns signals reversible erasures. We will proceed on a clause by clause basis indexed by event:

OMEGA: At this point the (forward) computation is complete since the machine is empty except for the result value v on the value stack. The history h contains sufficient information to reverse the computation. The eval function writes OMEGA onto h to signify termination.

On detecting OMEGA, the inverse evalRev clause removes it (reversibly—it is a constant and was written on known empty storage) from the history without modifying any other machine state.

RESTORE: When eval detects an empty control list, a single value v on the stack, and a triple on the dump d, it replaces the current environment with e′ from the dump, prepends v to the dump's stack s′, makes this list the new value stack, and places the dump's control list c′ as the new control list. Note that the current environment e is replaced with e′ and that the current stack s is replaced with v::s′. Furthermore, the dump's head element is removed in this operation. Therefore, the RESTORE event carries the triple (s,e,d). This triple contains all components (and hence information) available in this clause's input argument.

The inverse clause in evalRev uses the saved information held by RESTORE to restore the stack, environment, and dump. Note that no information is being erased since $v \subseteq s$, $s' \subseteq d$, $e' \subseteq e$, $c \subseteq d$, and $d' \subseteq d$.

LOOKUP: When an identifier i is at the head of the control list c, eval calls lookup to retrieve i's value from the environment e. As previously described, lookup is reversible via lookupRev and prepends ASSOC events to the history as necessary. LOOKUP carries the sub-environment e′ returned from lookup since this will be needed to subsequently reverse the previous lookup call; it also carries the control list so that the identifier term (ID i) is not lost and can be erased upon reversal. This clause propagates s, e, c, and d to the recursive call without information loss.

In the LOOKUP clause of evalRev, a call to lookupRev first reconstitutes the identifier i that was looked up along with the environment e″ used for the lookup. The call to lookupRev removes any ASSOC events from the head of the history h; lookupRev returns the residual history as h′. Note that the equality $e = e''$ holds in this clause. No information is erased when moving in this direction since $v \subseteq e$ (in particular, it is in the head element of e′ passed to lookupRev and can therefore be erased), and $c' \subseteq c$.

MKCLOS: Evaluation of a lambda-abstraction (LAMBDA) with eval creates a closure value (CLOS) on the value stack. The only candidate for erasure in this case is the LAMBDA term itself at the head of the control list. If we were not to retain it in the history using the MKCLOS constructor, we could have evalRev construct a structurally similar version using the components of CLOS; however, the original datum for the original LAMBDA term would then still exist without being accessible and hence not reversibly erasable. Instead, MKCLOS therefore carries the original control list in the history.

In the inverse clause, evalRev erases the closure value from the stack and restores the control list from the one carried by MKCLOS in the history. Erasure of the closure value is reversible since it was allocated on blank storage by the corresponding clause in eval and $e' = e$, $t \subseteq c$, and $i \subseteq c$.

APPCLOS: Closure application in eval saves the current stack s′ (without the closure and the value v at the

head), the environment e, and the tail of the control stack c′ on the dump. The current environment is set to the environment e′ in the closure extended by binding variable i to v. The new control list is the term t in the closure. Here, we must additionally save the closure value and the original control list; hence, APPCLOS carries the input stack s and control list c. The environment e has been saved on the dump; this obviates it being carried in the history.

In reverse, evalRev restores the stack s and control list c from the history. The head triple on the dump is reversibly erased—this is possible since $s' \subseteq s$, and $c' \subseteq c$. The control term t may also be erased ($t \subseteq s$). Furthermore, all other input components propagate to the recursive call: $e' \subseteq e$.

MKSUSP: This clause is similar to eval's clause for MKCLOS (due to the similarity of closures and suspensions). Evaluation of an application (APP) with eval creates a suspension value (SUSP) on the value stack. The only candidate for erasure in this case is the APP term at the head of the control list. If we were not to retain it in the history using the MKSUSP constructor, we could still have evalRev construct a structurally similar version using the components of SUSP and the function f that eval places at the head of the control list. However, the original datum for the original APP term would then still exist without being accessible and hence not reversibly erasable. Therefore, MKSUSP carries the original control list in the history.

In the inverse clause, evalRev erases the suspension value from the stack and restores the control list from the one carried by MKSUSP in the history. Erasure of the suspension value is reversible since it was allocated on blank storage by the corresponding clause in eval and $e' = e$, $v \subseteq c$, and $i \subseteq c$. Furthermore, $f \subseteq c$ and MARK is a constant.

APPSUSP: This clause is similar to the APPCLOS clause. Suspension application in eval saves the current stack (except for the suspension at the head) s′, the environment e, and the control stack c on the dump. The current environment is set to the environment e′ in the closure. The new control list is the term t in the closure. Here, we must additionally save the suspension value; hence, APPSUSP carries the input stack s. The environment e has been saved on the dump; this obviates it being carried in the history.

In reverse, evalRev restores the stack s from the history. The head triple on the dump is reversibly erased—this is possible since $s' \subseteq s$. The control term t may also be erased ($t \subseteq s$). Furthermore, all inputs propagate to the recursive call: $e' \subseteq s$.

4 Discussion

Here we discuss two issues: extending SECD-H to primitive data types and staging SECD-H computations in order to trade space for time.

The pure SECD and SECD-H machines require the encoding of basic types (*e.g.*, booleans, integers, lists, *etc.*) as lambda terms. Such encodings, however, are cumbersome. Furthermore, evaluation of such encodings may adversely impact space usage on the history tape. It is straight forward to extend SECD-H with primitive base types. Additionally, one must supply reversible primitive functions that operate on a primitive type. Inclusion of an integer type, for example, would require reversible primitive functions that operate on integers (addition, multiplication, negation, *etc.*). Reversible primitive functions can introduce additional history events when necessary. Integer addition might, for example, retain an addend on the history; given a sum and single addend, SECD-H can then recover the other addend. Other primitive types can be integrated similarly.

Since we consider finite implementations of SECD-H space usage is critical. Furthermore, history tape usage is proportional to computation steps. Bennett[3] describes a technique for trading history space for time in his reversible Turing machine. This technique is also amenable to SECD-H computations and works as follows.

A computation on an initial value V_0 is *staged* into n subcomputations. A subcomputation is initiated on an empty history tape and completes when the space on the history tape is exhausted. Two SECD-H machines M_1 and M_2 holding the initial SECD state V_0 are used. In general, one machine (say M_1) is in a reset state (SECD state is V_0) when the other (say M_2) completes computation of stage i. The latter machine M_2 contains intermediate SECD states $V_0 \ldots V_i$. States $V_1 \ldots V_i$ are copied from M_2 to M_1, M_2 is reset by "undoing" each subcomputation in descending order from V_i to V_1. Such an "undo" operation involves computing forward from a V_j to V_{j+1} to generate a history with which a V_j can be removed. Computation now reverts to M_1. When M_1's history fills, this process repeats with M_2 as the reset machine and M_1 holding the intermediary SECD states. In this manner, history tape space is traded for time. Further details and analyses of this technique can be found in Bennett's article[3]. Note that this technique must preserve intermediate SECD states—SECD-H provides a mechanism for the empirical study of the sizes of such states.

5 Conclusions

SECD-H is an evaluator for the lambda calculus that is logically reversible; in particular, it never irreversibly erases information. As such, it is potentially a candidate

for low-power computation. It is derived from Landin's SECD evaluator[8] using Bennett's construction for reversible Turing machines[3]. SECD-H can reversibly evaluate programs written in functional languages since the lambda calculus serves as a compilation model for functional languages.

Future work includes evaluators for other lambda calculus evaluation schemes (e.g., call-by-value). Most immediate, however, is the direct inclusion of primitive data types such as booleans, integers, and lists in our evaluator—values of such types currently require an encoding as pure lambda-calculus terms. SECD-H can also be used in experiments with real programs to measure, for example, resource requirements for intermediate states and for the history tape.

Acknowledgement

Thanks to Andrew Appel for comments on an early version of this paper.

References

[1] Appel, A., and D. MacQueen, "A Standard ML compiler," *Functional Programming Languages and Computer Architecture* **274** (1987), 301–324.

[2] Barendregt, H. P., *The lambda calculus—its syntax and semantics*, North Holland (1984).

[3] Bennett, C. H., "Logical reversibility of computation," *IBM Journal of Research and Development* **17** (1973), 525–532.

[4] Bennett, C. H., "The thermodynamics of computation—a review," *International Journal of Theoretical Physics* **21:12** (1982), 905–940.

[5] Field, A. J., and P. G. Harrison, *Functional programming*, Addison–Wesley (1988).

[6] Hudak, P., S. P. Jones, and P. L. Wadler, "Report on the programming language Haskell: a non-strict, purely functional language, version 1.1," *Tech. Rep. no. YALEU/DCS/RR-777*, Yale University, Computer Science Dept. (Aug. 1991)

[7] Landauer, R., "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development* **3** (1961), 183–191.

[8] Landin, P. J., "The mechanical evaluation of expressions," *Computer Journal* **6** (1964), 308–320.

[9] McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine," *Communications of the ACM* (1960), 184–195.

[10] Milner, R., "A theory of type polymorphism in programming," *Journal of Computer and System Sciences* (1978), 348–375.

[11] Milner, R., M. Tofte, and R. Harper, *The definition of Standard ML*, MIT Press (1990).

[12] Paulson, L. C., *ML for the working programmer*, Cambridge University Press (1991).