Interleaving is Possible with Refined Abstract Machines: A New Approach to Engineering a Compiler

Steven E. Ganz
Indiana University

Daniel P. Friedman
Indiana University

We study abstract machines as a perspective from which to better approach the development of complex systems. We believe that abstract machines provide a form of modularity that has been largely under-appreciated, and that has the potential to yield varied and substantial benefits. An important notion in exploring the interaction of abstract machines is one of refinement of one machine by another, which provides the ability for processes, threads, programs, or even single instructions from several abstract machines to share processors and data.

1 Introduction

It is common advice that one should develop sofware in small steps, from working systems to working systems that become better and better approximations of the final goal. This ought to be particularly true of complex systems such as compilers. Compilers are often built in multiple passes and these reflect to varying degrees what we will call a transformational style, i.e., with each pass representing a transformation between programs with a well-defined semantics [8, 2, 9]. Rarely, however, can the changes be applied at a fine level of granularity such as one language construct at a time, or even one occurrence of a language construct at a time. Many such transformations, most notably a rewriting in continuation-passing style, are difficult or impossible to apply incrementally.

The intermediate points through which a language passes as compiler transformations are applied correspond to abstract machines. Programs at those points can either continue with successive compiler passes, or they can be executed directly by an implementation of the abstract machine. If a compiler is to be developed incrementally, it should certainly be possible to execute the program between passes. We might also hope that we could execute a program that contained some features of the source language and some features of the target language, by interleaving instructions to the abstract machines involved. We believe that exploration of the necessary and sufficient conditions for interleaving execution of multiple abstract machines, and of techniques that could be relevant in accomplishing interleaving would be useful, and we begin that exploration with this paper.

Normally, an expression in one language is interpreted with respect to a machine for that language. If a target machine architecture were to receive a source language instruction, one option would be for it to immediately compile that instruction to target machine code. It is also possible, though, to interpret source language instructions with respect to a target machine, so that the source language instruction has the same effect on the target machine as the sequence of target language instructions to which the source instruction would compile. Each instruction can be viewed as having a "fetch" phase in which storage is accessed, a "process" phase in which data is manipulated, and a "store" phase in which storage is updated. To process a source language instruction on a target-language machine, apply a data coersion function in one direction between fetching and processing, and one in the opposite direction between processing and storing. More generally, it is possible to interpret source language forms occurring within a target language program (and containing embedded forms of either language) with respect to a target machine. It is not in general possible to go in the opposite direction—to interpret target language forms with respect to a source language machine. When the latter is possible, we have what is known as a reflection [14].

Many difficulties develop in interpreting source language instructions with respect to a target language machine; these make the problem non-trivial. First, the indexing of memory may be different. Also, several data structures in the source language may be represented under a single data structure in the target language. Extending to multiple passes, we end up with a tree of data representations whose arcs represent pairs of data conversion procedures. The target-to-source conversion procedures must be called on the inputs to the source language instruction and the source-to-target conversion procedures must be called on the result, before it is stored.

Although it adds complexity to the implementation, the advantages of allowing interleaving are substantial. Where transformational compilers allow us to slice the task of building a compiler into subtasks in one dimension (source to target), interleaving allows us to do so along another dimension (by feature). Language forms can be translated individually in each pass, relying on the ability of the implementation to handle those not yet converted, so the compiler can be tested after much less extensive modifications to the program.

The motivation for this paper came from our experience in teaching compilers to undergraduates. We felt that this approach to presenting a compiler led to a program that was easier to understand, because each pass of the compiler generated code

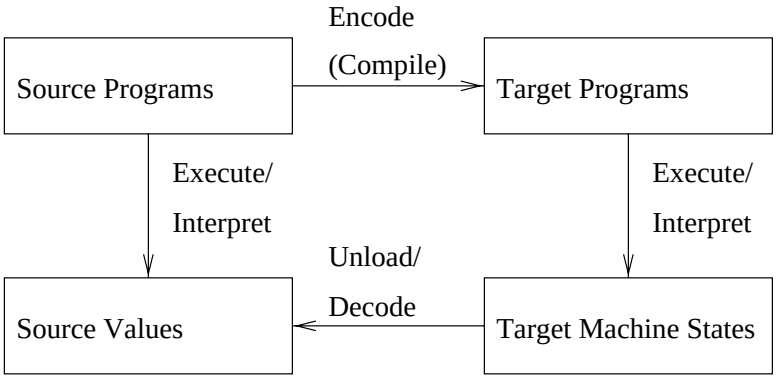

Figure 1: Burstall and Landin model of compilation in abstract algebra.

to implement a specific feature of its source language, in terms of only slightly lower-level features of its target language. It also led to programs that were easier to test and debug, since only small parts of the compiler, and of the compiled program, have changed since the last successful execution. The ongoing interest in proving compiler correctness [3, 11, 12, 16, 6] notwithstanding, we believe that this is still an important issue.

The rest of the paper proceeds as follows: We first lay the foundation for our perspective. The algebraic approach to compiler development has a rich history; we show how our results can be interpreted in that framework. Then, we present an example defining several abstract machines and show how they reduce the cost and complexity of developing compilers.

2 Foundations

Burstall and Landin [3] showed that compilation can be analyzed in terms of abstract algebras. Programs in the source and target languages each form the carrier of a term (word) algebra, whose operations are term constructors. Although the signatures of these algebras will most probably differ, the target machine algebra can always be restricted to an algebra with the source's signature. Algebra A is a restriction of algebra B if they share a carrier, and A's operations are derived from, i.e., statable in terms of, B's operations[1]. In this case, in the restricted algebra we can build up target language programs (possible results of compilation) by applying source language constructors. Any compiler is a homomorphism between the term algebra of the source language and the restricted term algebra of the target language. Values in the source language also form the carrier of an algebra with the same signature, and interpretation is a homomorphism between the term and value algebras of the source language. States of the target machine form the carrier of yet another algebra, which again can be restricted to one sharing the source's signature. Execution on the target machine is a homomorphism between the target machine's term algebra and the target machine's state algebra. By a lemma, that function is also a homomorphism between the restricted algebras. The final homomorphism, called unload, is from restricted states of the target machine to source values. The compiler is correct iff interpretation of the source expression is equal to the composition[2] of compilation, execution on the target machine, and unloading. Interpretation of the source language and execution on the target machine correspond to natural semantics of the source and target languages. Morris [12] prefers the use of denotational semantics. He renames values and states as meanings and unload as decode, with similar effects. This leads to the diagram in Figure 1, and the stated requirement for compiler correctness amounts to a requirement that the diagram commute. Note that only algebras sharing the source signature are included in this and succeeding diagrams.

The first thing to notice about this approach is that it conflates various functions. This provides simpler diagrams at the expense of precision. Execution is not really a function from programs to machine states. Rather, what was called execution is the composition of loading and true execution. Here, load is used differently from in [3] and more in line with standard usage. Similarly, what was called unloading is the composition of true unloading of the value from the target machine, and decoding of the target machine value as a source machine value. Rather than full execution, we are interested in only a single step of execution, because we will want to switch machines between steps. This leads to the use of a reduction (small step) semantics in place of natural (big step) semantics. Additionally, states of the source machine are not included at all. Interpretation of the source program may require loading of the program onto a source machine, execution of the program on the source machine, and unloading of the result. Finally, there could be any number of stages of compilation and intermediate machines, rather than a single transition from a source to a target. All of this leads us to the diagram in Figure 2. This diagram is the basis for the code that follows.

As an aside, we consider simplifications to the diagram in Figure 2. Since the state of the machine can (naturally) be considered to include the stored program in memory as well as a program counter or other measure of execution progress, we do not need to consider compilation of programs separately from the encoding of an entire machine state in another. The encoding function could be defined between machine states, rather than between programs. We might have hoped to also modify the decode function to map between machine states, and this is indeed possible for injective state encodings. Such a

[1] B's operations may be composed with each other, and applied to constants.

[2] For clarity, operations are presented in the order in which they are to be applied.

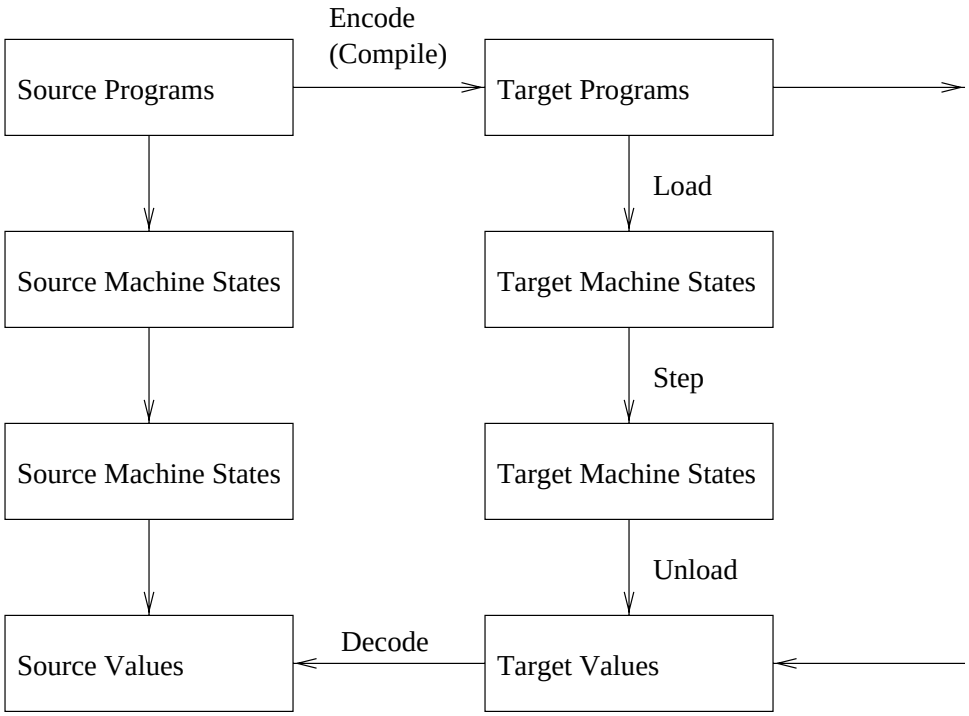

Figure 2: A model of compilation with program encoding and value decoding.

decode function is similar to what is referred to as a refinement function by Abadi and Lamport [1].[3] This perspective leads to the diagram in Figure 3.

Recall that natural (big step) semantics, or full execution, is a function between machine states. When two machines satisfy the above requirement for compiler correctness (of a commuting diagram) in this context, we say that the target machine simulates the source, or that the source machine reduces to the target [15]. It would be more demanding to require that the diagram commute when reduction (small step) semantics are used. Then, the target machine would have to relate back to the source machine at small intervals of execution.

State-encoding functions, however, do not reflect any actual execution in a computer system. Additionally, an injective state encoding may not be a reasonable requirement to impose (although Abadi and Lamport [1] contrive machine modifications that make them more applicable). It is often convenient to allow reuse of representations in the target for various source features. Even if it were possible to tease them apart by context, the result would be an unnecessarily complicated decoding function.

All of this may seem somewhat pedantic, so we get to the reason for the added precision. We want to allow for *source machine fragments mixed into the target machine state.* Although there are other possiblilities, we are primarily interested in source machine program fragments mixed into the target machine program. What would it mean to execute source-machine program fragments on the target machine? In the general case, it would mean that redexes whose top-level structure is recognized by the source machine could be used in place of redexes whose top-level structure is recognized by the target machine, within the program running on the target machine. In the case of a machine with a sequential instruction stream, this is just to say that source and target instructions can be interleaved in the instruction stream on the target machine. If we needed to execute source-machine program fragments on the target machine, how could this be accomplished? One method would be to compile away the source-like aspects of the program as they are encountered by the target machine. Another is to simply use the source-machine interpreter, but filter all accesses and updates to part of the source machine's state through the local encoding of the source machine in the target machine. Since from the target machine's perspective, source-machine distinctions may not be relevant, we let the source machine guide the decoding by presenting the kind of data that it expects. For example, if a source machine instruction running on the target machine requests the value of the accumulator (which holds a typed value), it requests that the target machine decode the corresponding register bits as including a type-tag. This latter technique is used below. This gives us the commuting diagram in Figure 4.

[3] There are several differences between Abadi and Lamport's formulation and that described here. First, to model output at every step, rather than just at the end of the computation, they distinguish a part of the state as being externally visible, and ensure that it is shared between the source and target. By assumption, it is only the externally visible component that need be preserved. Also, their formulation is time-based, so they must consider "steps" in which nothing happens to the state of a machine.

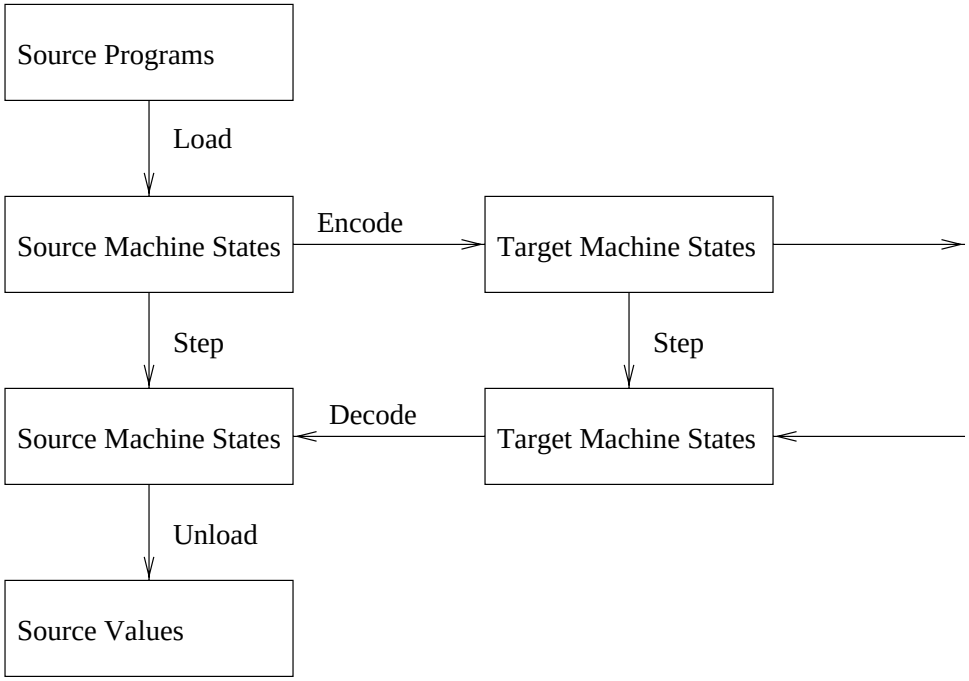

Figure 3: A model of compilation with state encoding and decoding.

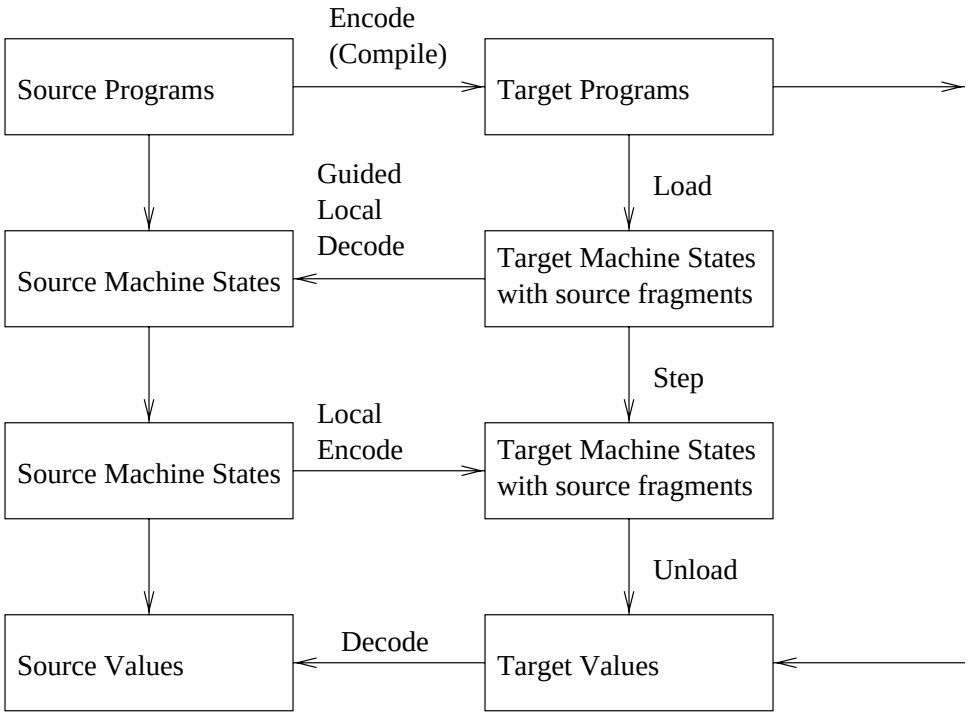

Figure 4: A model of compilation with local encoding and decoding.

3 Extended Example

As an extended example of the use of abstract machines to promote incremental software development and verification, we describe part of a Scheme compiler that we have implemented. The compiler is written in a transformational style, as described above. Two intermediate points in the transformation process are highlighted. We present implementations of each machine, and compilers that perform the transformations.

Programs are simply lists of instructions and labels. The format of instructions and labels will be described below. Instructions can be for either of the two intermediate machines or for the final target machine, and must contain the "quit" instruction (described below). The machines have substantial differences of a global nature that make this ability to interleave instructions for various machines non-trivial. We believe that this approach could be extended to cover numerous other intermediate machines.

3.1 Informal Descriptions

The two intermediate machines that we have selected are the Scheme Machine (SM) and the Symbolic Data Alpha Machine (SDAM). Our target machine is the Alpha Machine (AM). The SM uses only Scheme data, and its operations refer to Scheme entities such as closures. Scheme is relied on for primitive calls and heap management. The SDAM distinguishes between typed (tagged) and untyped Scheme data, and includes operations for adding and removing tags, in addition to operations common to assembly language, although at a slightly higher level and with vastly more developed data structures. The target machine interprets a parenthesized variant of DEC Alpha assembly code. It uses byte-addressable memory and a byte-field representation for most data. Both the SM and SDAM use word-addressable memory, i.e., each memory cell or register contain a single datum.

3.1.1 Scheme Machine

The SM is an intermediate-level abstract machine for Scheme, i.e., it models a run-time architecture for Scheme at an intermediate level of abstraction. It can be compared with several abstract machines previously presented [4, 5]. It operates directly over only frame indexes, labels, primitive names and trivial data references. The frame, frame-pointer, closures, closure pointer and accumulator are implicit but integral to the machine's operation. Only Scheme data and labels can be manipulated. Scheme data is represented as itself. Labels are represented by the list of instructions beginning with the declaration of that label. Lists are also Scheme data, so to maintain the injectivity of the local encoding (if we implement symbols), we `cons` a unique tag onto the list of instructions. The SM makes no reference to heap allocation of memory or to data types. Scheme primitives may be called freely.

The memory is now used only to hold stack data. Only three registers need be represented: the accumulator (ac), the frame pointer (fp) and the closure pointer (cp). Closures are represented as a vector whose first element is a label and whose remaining elements are a sequence of values.

3.1.2 Symbolic Data Alpha Machine

The Symbolic Data Alpha Machine (SDAM) is another abstract machine, residing at a somewhat lower level of abstraction than the Scheme Machine. It allows typed Scheme data to be manipulated, and also untyped data. The instructions, however, are similar to those on the Alpha Machine. It operates directly over registers and offsets. There is no longer any reference to closures. The accumulator, frame-pointer, and closure pointer are just registers; instructions do not distinguish among them (although both instructions and registers are "typed", i.e., expect specific kinds of data). Heap allocation must be performed by the user through the use of the allocation-pointer. This is required for the implementation of closures, `cons` cells, and other aggregate or mutable first-class data types. Scheme primitives are no longer available. Instructions are available to add type tags to and remove them from data.

The kinds of data encountered by the SDAM are as follows: various simple Scheme data (including the empty list (`null`), `boolean` values, `char`acters, and `integer`s), `type`s, `index`es, type-tagged Scheme data (`tdata`), and `label`s.

The representations are unique, i.e., one can determine the kind of data just by analyzing it. This property will not be maintained with the AM below, however. Untagged simple Scheme data is represented directly as Scheme data. Types are represented as symbols. Tagged scheme data is represened as a symbol composed of the value and the type, separated by a colon, and preceded by an "@". In the case of heap data, the value is a heap index.

Our system implemented the following types: the empty list, booleans, characters, integers, boxes, pairs, vectors, strings and procedures. The first three types share a type tag of "finite" to keep the number of types representable within three bits. Thus, the list above is actually of the extended types. We omit here presentation of boxes, vectors and strings due to space constraints.

SDAM has 7 registers available, dedicated to specific kinds of data. The accumulator (ac) holds typed data. The closure pointer (cp) holds a closure, also typed. The frame pointer (fp) and allocation pointer (ap) hold the indexes in memory of the beginning of the current frame and current closure, respectively. There are also three general-purpose registers (t1, t2 and t3).

Memory must now be used for both a stack and a heap. No care is taken to avoid overflows from the stack into the heap or from the heap out of memory. Our implementation deals with this through interrupts.

3.1.3 Alpha Machine

With the DEC Alpha Machine (AM), we reach the level of abstraction implemented in hardware. Our Alpha Machine processes a language similar but not identical to the assembly language of the original. There are many syntactic differences, and we use an abbreviated instruction set. The main difference, however, is in the way labels stored in memory are represented.

There are two predominant differences between the SDAM and the AM. First, the Alpha uses byte addressing of memory. It is not possible to hold the information of an SDAM datum in a single byte. A quadword will thus be required (quadwords contain eight bytes). This leads to a second difference—the quadword is the primary data type on the AM. Quadwords are represented as a list of 8 numbers ranging from 0 to 255. We call this a byte-field representation. Most data must be fit into this representation. For typed data, this is handled by allotting the three least significant bits as a type tag. Heap addresses would lose those three bits, but since memory is byte addressable and our data fall on quadword boundaries, we can regain a heap address by clearing the type tag. Our simulated alpha has a second datatype for labels, which are represented in the same way as on the SM and SDAM. All of the kinds of data known by the SDAM will have to be translated into quadword representations. Another difference is that the AM has more registers, and that they are referred to as numerals preceded by a "$".

3.2 Interleaving

These machines could be implemented independently, with relative ease. We choose to implement them in such a way that we can interleave instructions for various machines in a single program.

In all of our machines, a state is composed of registers, represented as a function from register names and kinds of data to values, as well as memory, represented as a function from positions and kinds of data to values, and the instruction stream, represented as a list. The instruction stream acts as a code pointer, but allows for the (unused) potential of self-modifying code.

An abstraction relation relates a state of a machine to a state of a relatively more abstract machine.

Is the abstraction relation injective? There is more than one way to compile source code, but our abstraction relation assumes a particular compiler. In relating AM to SDAM, there is a unique representation for every datum (potential value in a register or memory cell), so it is certainly injective.

Our abstraction relations are not functional, since different source can generate the same target code. Would things change if state did not include instructions, but merely a pointer? It might be possible to reconstruct the high-level configuration from the low-level one. What about cell-cell correlation? Relating AM to SDAM, there is no function from low-level values to high-level values. However, for any SDAM kind, there is a function from low-level values to high-level values of that kind.

There are fundamentally two ways to interleave instructions: running high-level instructions on a low-level machine and running low-level instructions on a high-level machine. The former occurs when a program is not fully compiled for the machine on which it is running. We can compile the high-level instruction and run the resulting low-level instructions, but we choose to run the high-level instruction directly. The latter way of interleaving instructions leaves open the question of whether the high level machine is itself implemented in the low-level machine. Cases where that is so arise with just-in-time compilation, and bring up issues of reflection.

The first issue to be confronted involves the fact that some states of the low-level machine, having less restrictive invariants, might violate constraints demanded by the high-level machine. Thus, when running high-level instructions on a low-level machine, it must be possible to verify that the low-level machine is in a "safe" state before running the high-level instruction. Running low-level instructions on a high-level machine brings up the issue that the low-level machine, might leave the system in a state that violates constraints imposed by higher-level machines, so that a high-level instruction cannot be performed. If this is the case, we must trust, or be able to verify, that the particular sequence of low-level instructions terminates with the machine in a "safe" state.

A second issue involves data conversions. Since the data representations differ between machines, it is not in general possible to execute instructions for any machine on an implementation of another machine. To surmount this, each instruction must coerce its inputs to their expected representation, and coerce its outputs to the representation required by the current implementation. This is reminiscent of the coersion between boxed and unboxed representations of data by Leroy [10]. In fact, one could view boxed and unboxed representations as belonging to separate abstract machines, and Leroy's technique as allowing instructions from those abstract machines to be interleaved.

Often, several data structures in the source language are represented under a single data structure in the target language. Extending to multiple passes, we end up with a tree of data representations whose arcs represent pairs of data conversion procedures. The AM supports two kinds of data, byte fields and labels. Labels are consistent across all machines, but byte fields represent the SDAM kinds: integers, indexes, typed data, types, characters, booleans and the empty list. Various types of typed data include integer, pair, procedure and finite. Each type of typed data represents SM data of that type.

Running high-level instructions on a low-level machine, it is necessary to convert data to be written to the implementation state to the lower-level format. This is straightforward for SDAM on AM because of the cell-cell correlation. It is also necessary to convert data read from the implementation state to the higher level format. SDAM instructions can be run on AM, if the SDAM instruction provides information regarding the kind of data expected.

The data conversion issue does not arise when running low-level instructions on a high-level machine implemented in the low-level machine. If implemented otherwise, we still need to perform data conversions, and have a situation which is parallel to that above.

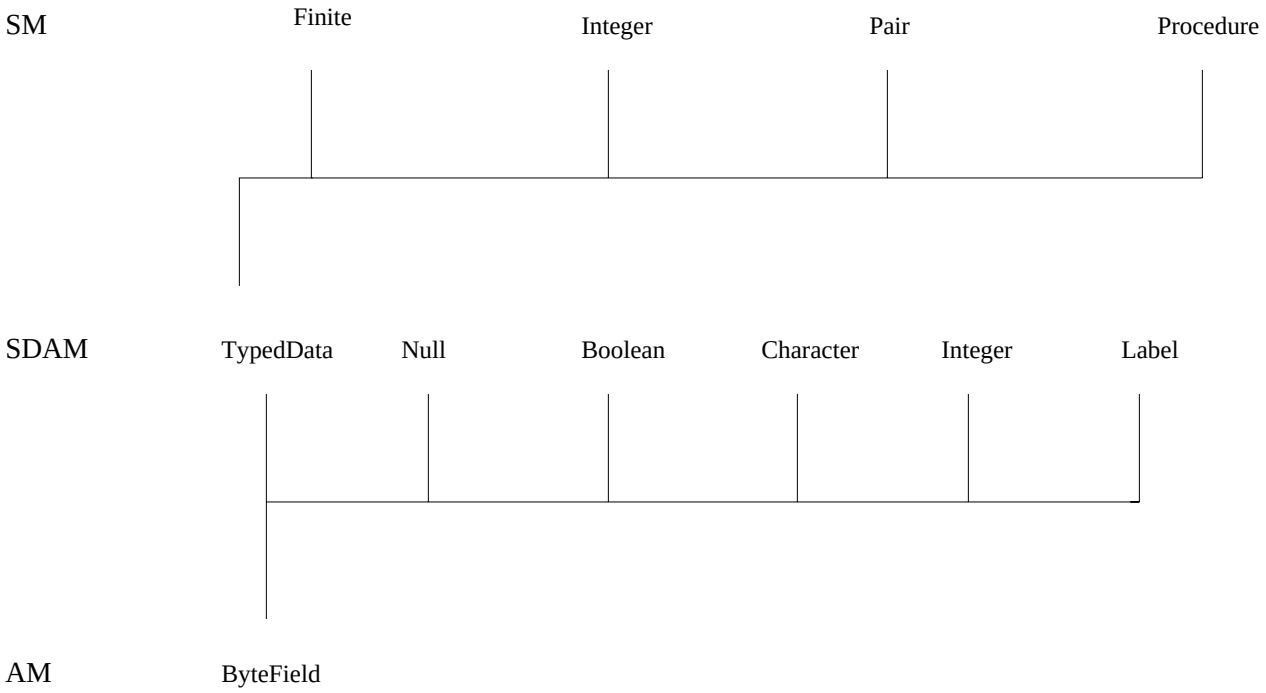

Figure 5: Relationships between kinds of data in the three machines. Data can be converted in either direction between SM and SDAM, but only from SDAM to AM.

The coersions described above are implemented by the functions representing registers and memory. `out-proc` is called on the result of reads. `in-proc` is called on the input to writes.

⟨ *Coersions* ⟩≡
```
(define extend
  (lambda (IMPLEMENTATION INSTRUCTION-SET)
    (lambda (REGISTERS MEMORY INSTR)
      (lambda (STRUCT KEY VAL KIND-IN)
        (let ([VAL (((in-proc REGISTERS MEMORY INSTR)
                     KIND-IN IMPLEMENTATION INSTRUCTION-SET)
                    VAL)])
             (lambda (X KIND-OUT)
               (if (eq? X KEY)
                   (((out-proc REGISTERS MEMORY INSTR)
                     KIND-OUT IMPLEMENTATION INSTRUCTION-SET)
                    VAL)
                   (STRUCT X KIND-OUT))))))))

  (define conversion-procs
    `((integer (,bf->integer ,integer->bf))
      (index (,bf->index ,index->bf))
      (tdata (,bf->tdata ,tdata->bf))
      (type (,bf->type ,type->bf))
      (label (,identity ,identity))
      (char (,bf->char ,char->bf))
      (boolean (,bf->boolean ,boolean->bf))
      (null (,bf->null ,null->bf))))
```

⟨ *Coersions* ⟩+≡
```
(define out-proc
  (lambda (REGISTERS MEMORY INSTR)
    (lambda (KIND IMPLEMENTATION INSTRUCTION-SET)
      (case IMPLEMENTATION
            [(AM)
             (case INSTRUCTION-SET
                   [(AM) identity]
                   [(SDAM) (car (cadr (assq KIND conversion-procs)))]
                   [(SM) (compose
```

```
                        (SM/SDAM-decode-value REGISTERS MEMORY INSTR)
                        (car (cadr (assq KIND conversion-procs))))])]
             [(SDAM)
              (case INSTRUCTION-SET
                    [(AM) (cadr (cadr (assq KIND conversion-procs)))]
                    [(SDAM) identity]
                    [(SM) (SM/SDAM-decode-value REGISTERS MEMORY INSTR)])]
             [(SM)
              (case INSTRUCTION-SET
                    [(AM) (compose encode (cadr (cadr (assq KIND conversion-procs))))]
                    [(SDAM) encode]
                    [(SM) identity])])))))

  (define in-proc
    (lambda (REGISTERS MEMORY INSTR)
      (lambda (KIND IMPLEMENTATION INSTRUCTION-SET)
        (case IMPLEMENTATION
              [(AM) (case INSTRUCTION-SET
                          [(AM) identity]
                          [(SDAM) (cadr (cadr (assq KIND conversion-procs)))]
                          [(SM)  (compose
                                   (cadr (cadr (assq KIND conversion-procs)))
                                   encode)])]
              [(SDAM)  (case INSTRUCTION-SET
                             [(AM) (car (cadr (assq KIND conversion-procs)))]
                             [(SDAM) identity]
                             [(SM) encode])]
              [(SM) (case INSTRUCTION-SET
                          [(AM) (compose
                                  (SM/SDAM-decode-value REGISTERS MEMORY INSTR)
                                  (car (cadr (assq KIND conversion-procs))))]
                          [(SDAM) (SM/SDAM-decode-value REGISTERS MEMORY INSTR)]
                          [(SM)  identity])])))))
```

To translate data from AM representation to SDAM representation, information regarding its kind is needed (it is used in both directions). There is no problem translating to or from SM, because the SM will only request data of kind `tdata` from the SDAM, and the SM can only offer typed data.

3.3 Formal Descriptions

The operational semantics of the machines will be presented as step procedures (written in an augmented variant of Scheme) that transform the state of the machine. This style is an extention of that used previously [13, 7, 4].

A pattern-matching notation is used in the machine descriptions below. `(syncase exp [pattern result] ...)` is evaluated as follows: `exp` is evaluated to some value `v`. `v` is matched against `pattern`. The only patterns supported are quasiquoted expressions and variables. Quoted portions of `pattern` must match corresponding portions of `v` exactly. (Sufficiently) unquoted variables are bound to corresponding portions of `v` for the evaluation of `result`. Subexpressions in the pattern followed by elipses must be matched by repeated items in `v`. References to variables from this portion of the pattern in `result` must be embedded in equally many elipses. When an elipses is applied to a pattern of the form `pattern:index-var`, `pattern` is matched against each repeated item in `v` and `index-var` is bound to the zero-based position of that item in the sequence. `(range-size index-var)` refers to the number of repeated items in the sequence indexed by `index-var`.

The following derived notations are also used:

```
(synlambda pattern body) →
   (lambda (x) (syncase x [pattern body]))
(synlet ([pattern rhs] ...) body) →
   ((synlambda `(,pattern ...) body) `(,rhs ...))
(synlet* () body) →
   (synlet () body)
(synlet* ([pattern1 rhs1] [pattern rhs] ...) body) →
   (synlet ([pattern1 rhs1]) (synlet* ([pattern rhs] ...) body))
```

For each machine, we will present a routine that loads a program into that machine (takes a program, returns a machine state), a routine that single-steps execution (takes and returns a machine state), and a routine that unloads a value (takes a machine state and returns a value). The value to be unloaded is always in the accumulator.

3.3.1 Scheme Machine

The SM is initialized with reasonable register values. The value in the fp is of type "index". Memory is initialized to 0.

⟨*SM*⟩≡
```
(define SM-load-program
  (lambda (PROG)
    (let ([REGISTERS (lambda (reg kind) 0)]
          [MEMORY (lambda (pos kind) 0)]
          [INSTR (cons linear-tag (cdr PROG))])
         (let ([extend ((extend 'SM 'SM) REGISTERS MEMORY INSTR)])
              `(,(extend
                   (extend
                     (extend REGISTERS
                         'fp 0 'index)
                     'cp (vector) 'tdata)
                   'ac 0 'tdata)
                ,MEMORY
                ,INSTR)))))

(define SM-step
  (lambda (IMPLEMENTATION)
    (lambda (REGISTERS MEMORY INSTR)
      (let ([extend ((extend IMPLEMENTATION 'SM) REGISTERS MEMORY INSTR)])
           (syncase (car INSTR)
               [`(label ,LAB)
                `(,REGISTERS ,MEMORY ,(cdr INSTR))]
               [`(SM-jump-to-label-from-curr-closure)
                `(,REGISTERS ,MEMORY ,(cdr (CP 0)))]
               [`(SM-load-curr-closure-from ,TRIV)
                `(,(extend REGISTERS 'cp (triv TRIV) 'tdata) ,MEMORY ,(cdr INSTR))]
               [`(SM-store-curr-closure-at-frame-pos ,POS)
                `(,REGISTERS ,(extend MEMORY (+ (REGISTERS 'fp 'index) POS)
                                      (REGISTERS 'cp 'tdata)
                                      'tdata)
                   ,(cdr INSTR))]
               [`(SM-make-closure-to-ac ,LAB ,TRIV ...)
                `(,(extend REGISTERS 'ac (list->vector `(,LAB ,(triv TRIV) ...)) 'tdata)
                  ,MEMORY ,(cdr INSTR))]
               [`(SM-call-primitive-to-ac ,PROC-NAME ,TRIV ...)
                `(,(extend REGISTERS 'ac (PROC-NAME (triv TRIV) ...) 'tdata)
                  ,MEMORY ,(cdr INSTR))]
               [`(quit) `(,REGISTERS ,MEMORY ,INSTR)])))))

(define linear-tag (gensym))
```

Trivial data references are of three types: local variable reference, free variable reference and literal. Each trivial data reference contains a tag identifying its type and a single argument. They are resolved as follows:

⟨*triv*⟩≡
```
(define triv
  (lambda (TRIV)
    (syncase TRIV
      [`(local* ,POS) (STACK (+ FP POS 1))]
      [`(free* ,POS) (vector-ref CP (+ POS 1))]
      [`(lit* ,LIT) LIT])))
```

Quite a few SM operations have been omitted, such as those adjusting the accumulator and frame pointer, and those storing data to and retrieving it from the current frame.

⟨*SM*⟩+≡
```
(define SM-unload-value
  (lambda (REGISTERS MEMORY INSTR)
    (REGISTERS 'ac)))
```

3.3.2 Symbolic Data Alpha Machine

In addition to the previous conventions, the following additional one will hold: `(arg-ref REGISTERS ARG KIND)` expands to `(if (register? ARG) (REGISTERS ARG KIND) ARG)`

This is because some SDAM (and AM) instructions take either a register name or a literal value.

Recall that the cp and ac take typed data on the SDAM.

⟨*SDAM*⟩≡

```
(define SDAM-load-program
  (lambda (PROG)
    (let ([REGISTERS (lambda (reg kind) 0)]
          [MEMORY (lambda (reg kind) 0)]
          [INSTR (cons linear-tag (cdr PROG))])
      (let ([extend ((extend 'SDAM 'SDAM)
                     REGISTERS MEMORY INSTR)])
        `(,(extend
        (extend
          (extend
            (extend REGISTERS
              'fp 0 'index)
            'ap 2000 'index)
          'cp (type 2000 'procedure) 'tdata)
        'ac (type 0 'integer) 'tdata)
       ,MEMORY
       ,INSTR)))))
```

The SDAM instructions and the corresponding state transitions are as follows:

⟨*SDAM*⟩+≡

```
(define SDAM-step
  (lambda (IMPLEMENTATION)
    (lambda (REGISTERS MEMORY INSTR)
      (let ([extend ((extend IMPLEMENTATION 'SDAM) REGISTERS MEMORY INSTR)])
        (syncase (car INSTR)
          [`(SDAM-jump ,DEST-REG)
           `(,REGISTERS ,MEMORY ,(cdr (REGISTERS DEST-REG 'label)))]
          [`(SDAM-load-address ,TO-REG ,LAB)
           `(,(extend REGISTERS TO-REG
                (cons linear-tag (member `(label ,LAB) (cdr INSTR)))
                'label)
              ,MEMORY ,(cdr INSTR))]
          [`(SDAM-load-immediate ,TO-REG ,VAL)
           `(,(extend REGISTERS TO-REG VAL (kind-of VAL))
             ,MEMORY ,(cdr INSTR))]
          [`(SDAM-load ,TO-REG ,FROM-BASE-REG ,FROM-DISP)
           `(,(extend REGISTERS TO-REG
                (MEMORY (+ (REGISTERS FROM-BASE-REG 'index) FROM-DISP) 'tdata)
                'tdata)
              ,MEMORY ,(cdr INSTR))]
          [`(SDAM-store ,FROM-REG ,TO-BASE-REG ,TO-DISP)
           `(,REGISTERS
              ,(extend MEMORY (+ (REGISTERS TO-BASE-REG 'index) TO-DISP)
                 (REGISTERS FROM-REG 'tdata) 'tdata)
              ,(cdr INSTR))]
          [`(SDAM-move ,FROM-REG ,TO-REG)
           `(,(extend REGISTERS TO-REG (REGISTERS FROM-REG 'any) 'any)
             ,MEMORY ,(cdr INSTR))]
          [`(SDAM-add-integer ,FROM-REG ,FROM-ARG ,TO-REG)
           `(,(extend REGISTERS TO-REG
                (+ (REGISTERS FROM-REG 'integer)
                  (arg-ref REGISTERS FROM-ARG 'integer))
                'integer)
              ,MEMORY ,(cdr INSTR))]
          [`(SDAM-eq? ,FROM-REG ,FROM-ARG ,TO-REG)
           `(,(extend REGISTERS TO-REG
                (equal? (REGISTERS FROM-REG 'any)
                  (arg-ref REGISTERS FROM-ARG 'any))
                'boolean)
              ,MEMORY ,(cdr INSTR))]
          [`(SDAM-add-address ,FROM-REG ,FROM-ARG ,TO-REG)
           `(,(extend REGISTERS TO-REG
                (+ (REGISTERS FROM-REG 'index)
                  (arg-ref REGISTERS FROM-ARG 'integer))
                'index)
              ,MEMORY ,(cdr INSTR))]
          [`(SDAM-type ,FROM-REG ,TYPE ,TO-REG)
           `(,(extend REGISTERS TO-REG
                (type (REGISTERS FROM-REG (type->kind TYPE) TYPE)
                  'tdata))
```

```
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-type-erase ,FROM-REG ,TO-REG)
     `(,(let* ([TDATA (REGISTERS FROM-REG 'tdata)]
               [KIND (type->kind (type-of TDATA))])
          (extend REGISTERS TO-REG (type-erase TDATA) KIND))
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-type-of ,FROM-REG ,TO-REG)
     `(,(extend REGISTERS TO-REG
          (type-of (REGISTERS FROM-REG 'tdata))
          'type)
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-type? ,FROM-REG ,TYPE ,TO-REG)
     `(,(extend REGISTERS TO-REG (type? (REGISTERS FROM-REG 'type) TYPE)
          'boolean)
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-extended-erase ,FROM-REG ,TO-REG)
     `(,(let* ([TDATA (REGISTERS FROM-REG 'tdata)]
               [KIND (type->kind (type-of TDATA))])
          (extend REGISTERS TO-REG (extended-erase TDATA) KIND))
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-extended-of ,FROM-REG ,TO-REG)
     `(,(extend REGISTERS TO-REG (extended-of (REGISTERS FROM-REG 'tdata))
          'type)
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-extended? ,FROM-REG ,EXTENDED-TYPE ,TO-REG)
     `(,(extend REGISTERS TO-REG
          (extended? (REGISTERS FROM-REG 'type) EXTENDED-TYPE)
          'boolean)
      ,MEMORY ,(cdr INSTR))]
    [`(SDAM-heap-erase ,FROM-REG ,TO-REG)
     `(,(extend REGISTERS TO-REG (type-erase (REGISTERS FROM-REG 'tdata))
          'index)
      ,MEMORY ,(cdr INSTR))]
    [else
      ((SM-step IMPLEMENTATION) REGISTERS MEMORY INSTR)]))))

(define kind-of   ; used only for immediate values and result of erasing type
  (lambda (SYMBOLIC-DATUM)
    (cond
      [(integer? SYMBOLIC-DATUM)   'integer]
      [(char? SYMBOLIC-DATUM)      'char]
      [(boolean? SYMBOLIC-DATUM)   'boolean]
      [(null? SYMBOLIC-DATUM)      'null]
      [(symbol? SYMBOLIC-DATUM)    'type])))


(define type->kind
  (lambda (TYPE)
    (case TYPE
          [(null:finite)    'null]
          [(boolean:finite) 'boolean]
          [(char:finite)    'char]
          [(integer)        'integer]
          [else             'index])))
```

`type`, `type-erase`, `type-of`, `type?`, `extended-erase`, `extended-of` and `extended?` are simple symbol-manipulation procedures whose definitions are omitted. `type` appends a type name to an untyped value, creating an typed object (of kind `tdata`). `type-erase` removes the type name from a typed object. `type-of` returns the type of a typed object. `type?` is just a sybol equality test of the contents of an input register with a type name.

`SDAM-eq?` and `SDAM-move` must use the special type `any` because no information about the kind of input is available or needed.

There are some redundant instructions here. The same instructions could have been used to add integers and indexes, for example, and heap-erase is identical to type-erase. This is intentional—the additional level of detail will be needed after translation to the alpha machine, where the representations of integers and indexes, and of tagged heap data and other tagged data, will diverge somewhat.

⟨*SDAM*⟩+≡

```
(define SDAM-unload-value
  (lambda (REGISTERS MEMORY INSTR)
    (REGISTERS 'ac)))
```

3.3.3 Alpha Machine

The definition of arg-ref is changed slightly so that byte fields are always returned. `(arg-ref REGISTERS ARG KIND)` expands to

```
(if (register? ARG)
(REGISTERS ARG KIND)
(any->bf ARG))
```

Registers are named with a numeral preceded by "$" on the AM. When we want to use the SDAM names, we must do conversions.

⟨*convert-reg*⟩≡

```
(define convert-reg
  (lambda (SDAM-reg-name)
    (reg-name (lookup-reg SDAM-reg-name))))

(define lookup-reg
  (lambda (SDAM-reg-name)
    (caddr (assq SDAM-reg-name kinded-registers))))
(define kinded-registers
  '((ac tdata $0) (ap index $11) (fp index $10)
    (lp label $8) (cp tdata $9)(t1 gp $5) (t2 gp $6) (t3 gp $7)))
```

Now, we must convert all values to byte fields during initialization. There is an additional register to be initialized, the read-only zero register.

⟨*AM*⟩≡

```
(define AM-load-program
  (lambda (PROG)
    (let ([REGISTERS (lambda (reg kind) (integer->bf 0))]
          [MEMORY (lambda (pos kind) (integer->bf 0))]
          [INSTR (cons linear-tag (cdr PROG))])
         (let ([extend ((extend 'AM 'AM)
                        REGISTERS MEMORY INSTR)])
              `(,(extend
                   (extend
                     (extend
                       (extend REGISTERS
                         (convert-reg 'fp) (index->bf 0) 'bf)
                       (convert-reg 'ap) (index->bf 2000) 'bf)
                     (convert-reg 'cp) (tdata->bf (type 2000 'procedure)) 'bf)
                   (convert-reg 'ac) (tdata->bf (type 0 'integer)) 'bf)
                ,MEMORY
                ,INSTR)))))
```

⟨*AM*⟩+≡

```
(define AM-step
  (lambda (IMPLEMENTATION)
    (lambda (REGISTERS MEMORY INSTR)
      (let ([extend ((extend IMPLEMENTATION 'AM) REGISTERS MEMORY INSTR)])
             (syncase (car INSTR)
               [`(AM-jump ,TO-REG ,DEST-REG)
                `(,(extend REGISTERS TO-REG (cons linear-tag (cdr INSTR)) 'label)
                  ,MEMORY ,(cdr (REGISTERS DEST-REG)))]
               [`(AM-load-address ,TO-REG ,LAB)
                `(,(extend REGISTERS TO-REG
                     (cons linear-tag (member `(label ,LAB) (cdr INSTR)))
                     'label)
                  ,MEMORY ,(cdr INSTR))]
               [`(AM-load-immediate ,TO-REG ,VAL)
                `(,(extend REGISTERS TO-REG (any->bf VAL) 'bf)
                  ,MEMORY ,(cdr INSTR))]
               [`(AM-load ,TO-REG ,FROM-BASE-REG ,FROM-DISP)
                `(,(extend REGISTERS TO-REG
                     (MEMORY (+ (bf->integer (REGISTERS FROM-BASE-REG 'bf))
                                FROM-DISP)
                       'bf)
                     'bf)
                  ,MEMORY ,(cdr INSTR))]
               [`(AM-store ,FROM-REG ,TO-REG ,DISP)
                `(,REGISTERS
                   ,(extend MEMORY (+ (bf->integer (REGISTERS TO-REG 'bf)) DISP)
                      (REGISTERS FROM-REG 'bf) 'bf)
                   ,(cdr INSTR))]
```

```
[`(AM-move ,FROM-REG ,TO-REG)
 `(,(extend REGISTERS TO-REG (REGISTERS FROM-REG 'bf) 'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-add ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG
      (bf+ (REGISTERS FROM-REG 'bf)
        (arg-ref REGISTERS FROM-ARG 'bf)) 'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-equal? ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG (bf= (REGISTERS FROM-REG 'bf)
                                  (arg-ref REGISTERS FROM-ARG 'bf))
      'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-bit-and ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG (bf-and (REGISTERS FROM-REG 'bf)
                                  (arg-ref REGISTERS FROM-ARG 'bf))
      'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-bit-or ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG (bf-or (REGISTERS FROM-REG 'bf)
                                  (arg-ref REGISTERS FROM-ARG 'bf))
      'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-shift-right-logical ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG
      (bf-shift-right (REGISTERS FROM-REG 'bf)
        (arg-ref REGISTERS FROM-ARG 'bf))
      'bf)
   ,MEMORY ,(cdr INSTR))]
[`(AM-shift-left-logical ,FROM-REG ,FROM-ARG ,TO-REG)
 `(,(extend REGISTERS TO-REG
      (bf-shift-left (REGISTERS FROM-REG 'bf)
        (arg-ref REGISTERS FROM-ARG 'bf))
      'bf)
   ,MEMORY ,(cdr INSTR))]
[else
  ((SDAM-step IMPLEMENTATION) REGISTERS MEMORY INSTR)))))))
```

The definitions of `bf-and`, `bf-or`, `bf-shift-left` and `bf-shift-right` are omitted due to space constraints.

The following routine is used to convert SDAM data of any kind to a byte field. The inverse operation cannot be defined.

⟨*SDAM/AM*⟩≡

```
(define any->bf
  (lambda (SDAM-DATUM)
    (cond
      [(null? SDAM-DATUM) (integer->bf 0)]
      [(boolean? SDAM-DATUM) (boolean->bf SDAM-DATUM)]
      [(character? SDAM-DATUM) (integer->bf (char->integer SDAM-DATUM))]
      [(integer? SDAM-DATUM) (integer->bf SDAM-DATUM)]
      [(label? SDAM-DATUM) SDAM-DATUM]
      [(symbol? SDAM-DATUM) (tdata->bf SDAM-DATUM)])))
```

The routines `integer->bf`, `boolean->bf` and `tdata->bf` encode data of various kinds as byte fields upon which operations such as `bf-and` and `bf-shift-left` are defined. They are omitted due to space constraints. `label?` simply checks for a pair whose car is `linear-tag`.

⟨*AM*⟩+≡

```
(define AM-unload-value
  (lambda (REGISTERS MEMORY INSTR)
    (REGISTERS (convert-reg 'ac))))
```

3.4 Compilers and Decoders

For each adjacent pair of machines, we will present a program encoder (compiler, takes a source language program, returns a target language program) and a value decoder (curried, takes a machine state, then a target language value, returns a source language value).

3.4.1 Scheme Machine to Symbolic Data Alpha Machine

Every SM Instruction expands into a finite sequence of SDAM Instructions. No SM Instruction expands into a loop. Furthermore, no SM Instruction generates any new labels or new gotos.

⟨*SM/SDAM*⟩≡
```
(define SM/SDAM-encode-program
  (lambda (SM-PROGRAM)
    (cons (car (SM-PROGRAM))
      (map
        (lambda (SM-INSTR)
          (syncase SM-INSTR
            [`(label ,LAB) `((label ,LAB))]
            [`(SM-jump-to-label-from-curr-closure)
             `((SDAM-heap-erase cp t1)
               (SDAM-load lp t1 0)
               (SDAM-jump lp))]
            [`(SM-load-curr-closure-from ,TRIV)
             `(,@(load-triv 'cp TRIV))]
            [`(SM-store-curr-closure-at-frame-pos ,POS)
             `((SDAM-store cp fp ,POS))]
            [`(SM-make-closure-to-ac ,PROC-NAME ,TRIV:I ...)
             `((SDAM-move ap t1)
               (SDAM-type t1 'procedure ac)
               (SDAM-load-address lp ,PROC-NAME)
               (SDAM-store lp ap 0)
               ,@(store-triv 'ap (+ I 1) TRIV 't1) ...
               (SDAM-add-address ap ,(add1 (range-size I)) ap))]
            [`(SM-call-primitive-to-ac ,PROC-NAME ,TRIV ...)
             (SM/SDAM-encode-primitive-instr (cdr SM-INSTR))]
            [`(quit) `((quit))]))
        (cdr SM-PROGRAM)))))

(define load-triv
  (lambda (TO-REG-NAME TRIV)
    (syncase TRIV
      [`(lit* ,LIT)
       `((SDAM-load-immediate ,TO-REG-NAME ,LIT)
         (SDAM-type ,to/reg-name  ',(cond
                                      [(null? LIT) 'null:finite]
                                      [(boolean? LIT) 'boolean:finite]
                                      [(char? LIT) 'char:finite]
                                      [(number? LIT) 'integer])
           ,TO-REG-NAME))]
      [`(local* ,POS)
       `((SDAM-load ,TO-REG-NAME fp ,POS))]
      [`(free* ,POS)
       `((SDAM-heap-erase cp ,TO-REG-NAME)
         (SDAM-load ,TO-REG-NAME ,TO-REG-NAME ,POS))])))

(define store-triv
  (lambda (TO-BASE TO-DISP TRIV TEMP)
    (syncase TRIV
      [`(lit* ,LIT)
       `((SDAM-load-immediate ,TEMP ,LIT)
         (SDAM-type ,TEMP  ',(cond
                               [(null? LIT) 'null:finite]
                               [(boolean? LIT) 'boolean:finite]
                               [(char? LIT) 'char:finite]
                               [(number? LIT) 'integer])
           ,TEMP)
         (SDAM-store ,TEMP ,TO-BASE ,TO-DISP))]
      [`(local* ,POS)
       `((SDAM-load ,TEMP fp ,POS)
         (SDAM-store ,TEMP ,TO-BASE ,TO-DISP))]
      [`(free* ,POS)
       `((SDAM-heap-erase cp ,TEMP)
         (SDAM-load ,TEMP ,TEMP ,POS)
         (SDAM-store ,TEMP ,TO-BASE ,TO-DISP))])))
```

Only a sampling of primitives are included, due to space constraints.

⟨*SM/SDAM*⟩+≡
```
(define SM/SDAM-encode-primitive-instr
  (lambda (SM-PRIM-CALL)
    (syncase SM-PRIM-CALL
      [('cons 'ARG1 'ARG2)
       `(,@(load-triv 't1 ARG1)
```

```
           ,@(load-triv 't2 ARG2)
           (SDAM-move ap t3)
           (SDAM-type t3 'pair ac)
           (SDAM-store t1 ap 0)
           (SDAM-store t2 ap 1)
           (SDAM-add-address ap 2 ap))]
         [('car 'ARG1)
          `(,@(load-triv 't1 ARG1)
           (SDAM-heap-erase t1 t1)
           (SDAM-load ac t1 0))]
         [('cdr 'ARG1)
          `(,@(load-triv 't1 ARG1)
           (SDAM-heap-erase t1 t1)
           (SDAM-load ac t1 1))]
         [('set-car! 'ARG1 'ARG2)
          `(,@(load-triv 't1 ARG1)
           ,@(load-triv 'ac ARG2)
           (SDAM-heap-erase t1 t1)
           (SDAM-store ac t1 0))]
         [('set-cdr! 'ARG1 'ARG2)
          `(,@(load-triv 't1 ARG1)
           ,@(load-triv 'ac ARG2)
           (SDAM-heap-erase t1 t1)
           (SDAM-store ac t1 1))]
         [('pair? 'ARG1)
          `(,@(load-triv 't1 ARG1)
           (SDAM-type-of t1 t2)
           (SDAM-type? t2 'pair t2)
           (SDAM-type t2 'boolean:finite ac))]
         [('null? 'ARG1)
          `(,@(load-triv 't1 ARG1)
           (SDAM-extended-of t1 t2)
           (SDAM-extended? t2 'null:finite t2)
           (SDAM-type t2 'boolean:finite ac))])))
```

⟨*SM/SDAM*⟩+≡

```
(define SM/SDAM-decode-value
  (lambda (REGISTERS MEMORY INSTR)
    (lambda (TDATA)
      (case (typeof TDATA)
            [(finite)
             (extended-erase TDATA)]
            [(integer)
             (type-erase TDATA)]
            [(pair)
             (cons (decode (MEMORY (type-erase TDATA) 'tdata))
               (decode (MEMORY (add1 (type-erase TDATA)) 'tdata)))]
            [(procedure)
             (lambda (x) x)]))))
```

3.4.2 Symbolic Data Alpha Machine to Alpha Machine

⟨*SDAM/AM*⟩+≡

```
(define SDAM/AM-encode-program
  (lambda (SDAM-PROGRAM)
    (cons (car SDAM-PROGRAM)
      (map
        (lambda (SDAM-INSTR)
          (syncase SDAM-INSTR
            [(label LAB)
             `((label ,LAB))]
            [(SDAM-jump DEST-REG)
             `((AM-jump $1 ,(convert-reg DEST-REG)))]
            [(SDAM-load-address TO-REG LAB)
             `((AM-load-address ,(convert-reg TO-REG) ,LAB))]
            [(SDAM-load-immediate TO-REG VAL)
             `((AM-load-immediate ,(convert-reg TO-REG) ,(datum->integer VAL)))]
            [(SDAM-load TO-REG FROM-BASE FROM-DISP)
             `((AM-load ,(convert-reg TO-REG) ,(convert-reg FROM-BASE)
                 ,(* FROM-DISP 8)))]
```

```
            [(SDAM-store FROM-REG TO-BASE-REG TO-DISP)
             `((AM-store ,(convert-reg FROM-REG) ,(convert-reg TO-BASE-REG)
                 ,(* TO-DISP 8)))]
            [(SDAM-move FROM-REG TO-REG)
             `((AM-move ,(convert-reg FROM-REG) ,(convert-reg TO-REG)))]
            [(SDAM-add-integer FROM-REG FROM-ARG TO-REG)
             `((AM-add ,(convert-reg FROM-REG) ,(convert-reg-arg FROM-ARG)
                 ,(convert-reg TO-REG)))]
            [(SDAM-eq? FROM-REG FROM-ARG TO-REG)
             `((AM-equal? ,(convert-reg FROM-REG) ,(convert-reg-arg FROM-ARG)
                 ,(convert-reg TO-REG)))]
            [(SDAM-add-address FROM-REG FROM-ARG TO-REG)
             `((AM-shift-right-logical ,(convert-reg FROM-REG) 3 $1)
               (AM-add $1 ,(convert-reg-arg FROM-ARG) $1)
               (AM-shift-left-logical $1 3 ,(convert-reg TO-REG)))]
            [(SDAM-type FROM-REG TYPE TO-REG)
             (let ([ALPHA-TO-REG (convert-reg TO-REG)])
                  (if (memq TYPE '(procedure pair))
                      `((AM-bit-or ,(convert-reg FROM-REG)
                          ,(etype->integer type-name)
                          ,ALPHA-TO-REG))
                      `((AM-shift-left-logical ,(convert-reg FROM-REG)
                          ,(case type-name
                                 [(integer) 3]
                                 [(null:finite boolean:finite char:finite) 8])
                          ,ALPHA-TO-REG)
                        (AM-bit-or ,ALPHA-TO-REG ,(etype->integer type-name)
                          ,ALPHA-TO-REG))))]
            [(SDAM-type-erase FROM-REG TO-REG)
             `((AM-shift-right-logical ,(convert-reg FROM-REG) 3
                 ,(convert-reg TO-REG)))]
            [(SDAM-type-of FROM-REG TO-REG)
             (let ([ALPHA-TO-REG (convert-reg TO-REG)])
                  `((AM-extract-byte-logical ,(convert-reg FROM-REG) 0
                      ,ALPHA-TO-REG)
                    (AM-bit-and ,ALPHA-TO-REG 7 ,ALPHA-TO-REG)))]
            [(SDAM-type? FROM-REG TYPE-NAME TO-REG)
             `((AM-equal? ,(convert-reg FROM-REG) ,(type->integer TYPE-NAME)
                 ,(convert-reg TO-REG)))]
            [(SDAM-extended-erase FROM-REG TO-REG)
             `((AM-shift-right-logical ,(convert-reg FROM-REG) 8
                 ,(convert-reg TO-REG)))]
            [(SDAM-extended-of FROM-REG TO-REG)
             `((AM-extract-byte-logical ,(convert-reg FROM-REG) 0
                 ,(convert-reg TO-REG)))]
            [(SDAM-extended? FROM-REG TYPE-NAME TO-REG)
             `((AM-equal? ,(convert-reg FROM-REG) ,(etype->integer TYPE-NAME)
                 ,(convert-reg TO-REG)))]
            [(SDAM-heap-erase FROM-REG TO-REG)
             `((AM-bit-and ,(convert-reg FROM-REG) -8 ,(convert-reg TO-REG)))]
            [(quit) `((quit))]))
          (cdr PROGRAM)))))

  (define convert-reg-arg
    (lambda (SDAM-REG-ARG)
      (if (symbol? SDAM-REG-ARG)
          (convert-reg SDAM-REG-ARG)
          SDAM-REG-ARG)))
```

⟨*SDAM/AM*⟩+≡

```
  (define SDAM/AM-decode-value
    (lambda (REGISTERS MEMORY INSTR)
      (lambda (BF)
        (bf->tdata BF))))
```

4 Conclusions and Future Research

Every program is in some sense a machine, and every specification an abstract machine—each program must process specific requests in predefined ways. However, programming systems can choose to either encourage or shun this perspective. We have presented a demonstration of the benefits to be gained by treating abstract machines as a fundamental notion in the

design of a complex system. The compiler that we developed in this way is modular. Components can be reused in compilers with different source or target languages. It can be developed incrementally, since source and target language fragments can be interleaved. It has the additional benefit that for systems programming (such as of a garbage collector), one of the intermediate languages might be the right tool for the job. Code so written could be executed alongside the user's compiled code.

Interleaving is an approach to a problem that has been explored for the Java programming language. In the interests of portability, programs are distributed in the form of relatively high-level byte code designed to be run on the Java Virtual Machine. For frequently-run applications, however, there is an efficiency advantage in compiling byte code to native machine code, rather than processing it directly. Dual evaluation strategies are thus possible, corresponding to the options for our target machine upon receiving a source language expression. In this paper, we have considered the interpretation option, which may be appealing for code that is difficult to optimize or unlikely to be executed multiple times. Seeing the problem in an idealized form may help us to better appreciate and deal with it in practical situations such as this.

We hope that continuing to apply this approach would lead to more reliable compiler implementation through incremental program development and verification, as well as increased security in environments where users with varied access rights and language usage must interact. It is hoped that future research would provide guidelines for not just unregulated interleaving of instructions from various machines, but for some notion of safe interleaving. This, of course, leads into issues of parallel execution of multiple threads and processes. The simplest approach would be to treat the "evaluation steps" of any abstract machine as atomic operations.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302. ACM SIGACT and SIGPLAN, ACM Press, 1989.

[3] R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, volume 4, pages 17–43. University of Edinburgh Press, 1969.

[4] Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217. ACM, ACM, August 1984.

[5] William Clinger. The scheme 311 compiler: An exercise in denotational semantics. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, 1984.

[6] Joshua Guttman, John Ramsdell, and Mitchell Wand. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[7] Peter Henderson. *Functional Programming: Application and Implementation*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

[8] R. Kelsey and P. Hudak. Realistic compilation by program transformation (detailed summary). In ACM, editor, *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, pages 281–292, New York, NY, USA, 1989. ACM Press.

[9] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[10] Xavier Leroy. Unboxed objects, polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, January 1992.

[11] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic, 1972.

[12] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 144–152. ACM SIGACT and SIGPLAN, ACM Press, 1973.

[13] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Cambridge, Mass., 1981.

[14] Amr Sabry and Philip Wadler. A reflection on call-by-value. In *International Conference on Functional Programming '96*, pages 111–136. ACM Press, 1996.

[15] D. Scott. Some definitional suggestions for automata theory. *Journal of Computer and System Sciences*, 1:187–212, 1967.

[16] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. More on advice on structuring compilers and proving them correct. In Neil D. Jones, editor, *Proceedings of a Workshop on Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 165–188. Springer Verlag, 1980.