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

Steven E. Ganz, Daniel P. Friedman

Indiana University

We have studied 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 [6], [2], [7]. 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.

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 to implement a specific feature of the input language, with the output language of the pass beging only slightly lower-level than the input. 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 wide interest in proving compiler correctness [?] 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 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]. So we can find a way of building up target language programs that corresponds to applying source language operations. 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 decoding. Interpretation of the source language and execution on the target machine correspond to natural semantics of the source and target languages. Morris [9] prefers the use of denotational semantics. He renames values and states as meanings and unload as decode, with similar effects. The stated requirement for compiler correctness amounts to a requirement that compilation/execution 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

[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.

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.

As an aside, we consider simplifications. 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 decode function is similar to what is referred to as a refinement function by Abadi and Lamport [1].[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 [11]. 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*

[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.

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 1.

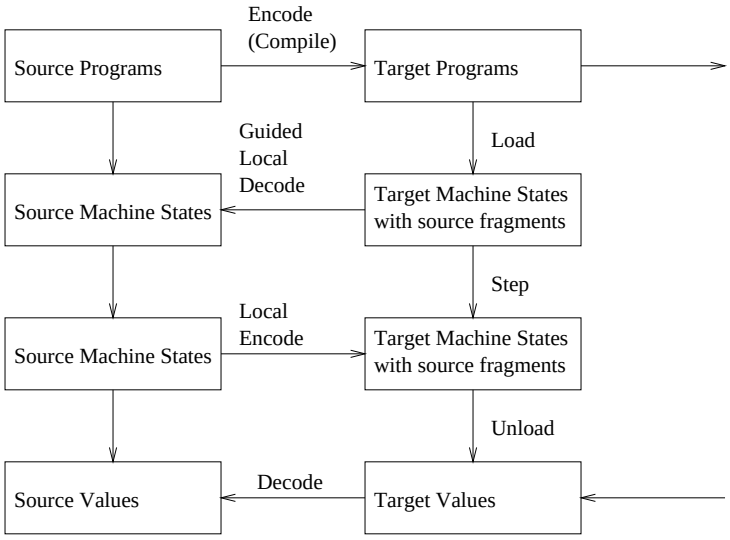

Fig. 1. A model of compilation with local encoding and decoding.

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.

Symbolic Data Alpha Machine The Symbolic Data Alpha Machine (SDAM) 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. 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. 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.

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 both 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 [8]. 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.

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 (STRUCT KEY VAL KIND-IN)
        (let ([VAL ((in-proc KIND-IN IMPLEMENTATION INSTRUCTION-SET)
                    VAL)])
             (lambda (X KIND-OUT)
               (if (eq? X KEY)
                 ((out-proc 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))))

(define out-proc
  (lambda (KIND IMPLEMENTATION INSTRUCTION-SET)
      (case IMPLEMENTATION
            [(AM)
```

```
             (case INSTRUCTION-SET
                   [(AM) identity]
                   [(SDAM) (car (cadr (assq KIND conversion-procs)))])]
            [(SDAM)
             (case INSTRUCTION-SET
                   [(AM) (cadr (cadr (assq KIND conversion-procs)))]
                   [(SDAM) identity])])))

(define in-proc
  (lambda (KIND IMPLEMENTATION INSTRUCTION-SET)
      (case IMPLEMENTATION
            [(AM) (case INSTRUCTION-SET
                        [(AM) identity]
                        [(SDAM) (cadr (cadr (assq KIND conversion-procs)))])]
            [(SDAM)  (case INSTRUCTION-SET
                           [(AM) (car (cadr (assq KIND conversion-procs)))]
                           [(SDAM) identity])])))
```

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 [10], [5], [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`. Symbols appearing in the pattern in upper case are bound to 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 corresponding elipses. Symbols of the form `ITEM-VAR:INDEX-VAR` in the pattern, within at least one level of elipses cause `ITEM-VAR` to be bound to each repeated item in `v` and `INDEX-VAR` to be bound to the zero-based position of that item. Multiple colons may be applied to bind positions within multiple elipses. `(range-size INDEX-VAR)` refers to the number of repeated items in the sequence indexed by `INDEX-VAR`.

The corresponding notations below are also used.

```
(synlet ([pattern rhs] ...) body)
(synlet* ([pattern rhs] ...) body)
(synlambda pattern 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.

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)))
```

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).

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.

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.

Another area of potential research is the alternative strategy of either statically or dynamically compiling source program fragments which appear in the target program.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. pages ???–???, ???, 1988.
2. Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. pages ???–???, Portland, Oregon, January 1989. ACM Press.
3. R. M. Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:???–???, 1969.
4. Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages ???–???, Austin, Texas, August 1984. ACM Press.
5. P. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall, ???, 1980.
6. Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. pages ???–???, Portland, Oregon, January 1989. ACM Press.
7. Richard Kelsey and Jonathan A. Rees. A tractable scheme implementation. *Lisp and Symbolic Computation*, ?:???–???, 1993.
8. Xavier Leroy. Unboxed objects and polymorphic typing. pages ???–???, Portland, Oregon, January 1992. ACM Press.
9. F. Lockwood Morris. Advice on structuring compilers and proving them correct. pages ???–???, Portland, Oregon, January 1973. ACM Press.
10. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Cambridge, Mass., 1981.
11. D. Scott. Some definitional suggestions for automata theory. *Journal of Computer and System Sciences*, 1:187–212, 1967.