Program Verification for Optimized Byte Copy

Edoardo S. Biagioni
July 1994
CMU-CS-94-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-94-06

Abstract

We present a program to copy bytes, together with a formal specification and a proof that the code satisfies the specification. The program, which is in the critical path for a network implementation, has been tuned carefully over a period of time; the proof covers the entire program, and is easily updated if the program is modified. The program is written in the Standard ML programming language and was produced as part of the Fox Project implementation of the TCP/IP protocol suite.

The author's electronic mail address is: esb@cs.cmu.edu

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.
**AD NUMBER**

**DATE**

**DTIC ACCESSION NOTICE**

<table>
<thead>
<tr>
<th>REPORT IDENTIFYING INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. ORIGINATING AGENCY</td>
</tr>
<tr>
<td>B. REPORT TITLE AND/OR NUMBER</td>
</tr>
<tr>
<td>C. MONITOR REPORT NUMBER</td>
</tr>
</tbody>
</table>

| PREPARED UNDER CONTRACT NUMBER | ARPA/STO 1962.8-91-2-0168 |

| DISTRIBUTION STATEMENT | (A) UNLIMITED |

**REQUESTER:**

1. Put your mailing address on reverse of form.
2. Complete Items 1 and 2.
3. Attach form to reports mailed to DTIC.
4. Use unclassified information only.
5. Do not order document for 6 to 8 weeks.

**DTIC:**

1. Assign AD Number.
2. Return to requester.
1 Introduction

Data copying is a common operation in systems code. Because it is frequently used, it is important that the implementation of data copying be optimized. Operating systems commonly provide a byte-copy library function that, for maximum performance, is written in assembly.

Assembly lets programmers optimize the performance of a specific algorithm, but higher-level languages let programmers easily try out different algorithms to discover which might be the most efficient. With sufficiently good compilers and sufficiently complicated architectures, a program in a more advanced language such as Standard ML (SML) or Lisp can even outperform one carefully written in a lower-level language [1], though for the copy code presented here we claim no such result\(^1\). One undesirable consequence of trying out different algorithms is the possibility of introducing errors. The main motivation for the proof presented in this paper was preventing the introduction of errors during optimization of a byte copy function written in SML.

The errors were discovered by Nevin Heintze when he applied set-based analysis \([5]\) to the code and noticed that the reported ranges for some of the array indices were outside the legal ranges. After verifying that the error was not in the set-based analysis code, the errors were shown to be in the copy code.

The discovery of such errors made us aware that more errors might be present. The byte copy code is imperative in style, and optimizations often obscure the meaning of the program; the copy code is therefore more prone to errors than most SML programs. Because of this, and because the module is easily specified, the author decided that a proof of correctness would be an effective strategy for increasing confidence in the correctness of the program.

The first proof was written in the pre-condition and post-condition style similar to that first suggested by Naur [8] and Floyd [3]. The proof was successful in that the resulting program, though exhaustively tested, showed no implementation bugs\(^2\). The inadequacy of this first proof became apparent, however, when it came time to further optimize the code: the proof was unmaintainable.

Further work by the author, with advice from Matthias Felleisen and Robert Harper, produced a transformational proof similar to those described by Mason [6]. The program has been modified since the proof was completed, and the proof modified with it, at little more cost in programmer time than it would take to describe and motivate in detail all the changes.

The author believes the significance of this work to lie in its being a practical and maintainable transformational proof of an optimized program. The transformational style of program proofs was selected in part because it is easily modified when the program is modified, leading to a maintainable proof.

The proof and the program were developed together; the author believes this strategy leads to simpler proofs and clearer and better programs than developing the program and attempting to prove it, or developing a proof and attempting to implement it.

The remainder of this paper describes the proof and the source code in greater detail. Section 2 describes some related research, Sections 3–13 give the specification, the source code, and the proof of the program, and Sections 14 and 15 summarize the work and point to some directions for further research.

\(^1\) However, our checksum code written in SML does outperform a production checksum code written in C which uses a less efficient algorithm.

\(^2\) The exhaustive testing did bring to light one specification error. Once the specification was amended and the code and proof changed correspondingly, the program tested correctly the first time it was run.
2 Background

2.1 The Fox Project

The Fox Project at CMU [2], is exploring the possibilities offered by an advanced programming language for implementing systems code. Currently the Fox Project is implementing networking protocols such as TCP and IP using the Standard ML (SML) language. SML offers an elaborate module system, polymorphic static type checking, and a notion of safety that guarantees that operations on one data value will not inadvertently affect other data values. The latter is not true in C, for example: an operation on a pointer which was initialized to point to one data value may well affect other, unrelated data values. This kind of safety is useful in preventing bugs whose manifestation is usually far-removed from the cause, and therefore hard to locate.

To maintain this kind of safety, as much as possible of the code is implemented in SML; this includes the byte copy code presented in this paper. As well as the safety property, characteristics of SML that are useful for proofs include the explicit identification of those objects whose value may change, a lack of explicit pointers, and the scoping introduced by modules.

At present (July 1994) we use a modified version of the SML/NJ compiler version 0.93; this compiler does not especially optimize loops over arrays. To obtain acceptable performance, we have performed by hand several classical optimizations such as loop unrolling, common-subexpression elimination, and case analysis. The success of this optimization can be seen by the fact that, on a typical system such as the DECstation 5000/120, the maximum performance for the highly optimized aligned copy, 3.4 megabytes per second, is over four times the 770 kilobytes per second of the unoptimized unaligned copy. This performance is similar to that of a simple C program that copies 4-byte words from one array to another, about 3.0 megabytes per second using the native C compiler (with -0 optimization), though much slower than that of a simple C program that copies 4-byte words between arrays by using pointers, about 23 megabytes per second, or of the library function bcopy, about 33 megabytes per second.

2.2 Proof Techniques

There are many proof techniques that are useful for proving algorithms. Of these, some are potentially useful for proving programs. With traditional pre-conditions and post-conditions, a program is proven by producing assertions that hold at different points in a program, and verifying that the pre-conditions together with the code can be used to derive the postconditions. More recent work by Scherlis and Scott introduces the idea that proofs should be maintainable; within this context Mason presents proof techniques suitable for building and maintaining proofs of destructive programs, that is, programs which are not purely functional. The technique presented by Mason is to prove that programs or program fragments are strongly isomorphic to one another, that is, they return the same value and have the same effect on the store; if two programs or program fragments are strongly isomorphic, then one can replace the other without affecting the meaning of the program. Though Lisp is the language used by Mason, the same techniques are applicable to SML programs. The proof presented in this paper uses all of these techniques.

The proof is divided into three parts. We first prove that the main function satisfies the specification, by showing that its specification is equivalent to that of its subsidiary functions and that exactly one of these functions is called by the main function.

We then prove that a base case, a simple, unoptimized routine, implements the specification. This proof uses traditional pre-condition and post-condition techniques and a counting argument equivalent to induction over the loop.
We finally transform the base case into the optimized routines using only simple, provably correct steps, with techniques similar to those described by Mason [6]. The correctness of the main function is derived from the specification of the other routines and the assumption that the routines are correct.

In all these proofs we assume that the program type checks, and that therefore there are no type errors. The proof itself is therefore largely typeless, and if fact loose in mixing 2-byte and 4-byte values.

2.3 Notation and Organization

The notation used in this paper is shown in Table 2.3. The code uses \texttt{andb} instead of the SML \texttt{mod} operation since SML/NJ 0.93 does not optimize modulo operations where the second operand is a constant power of two.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>program equivalence (strong isomorphism)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>andb(x, m-1)</td>
<td>(x \mod 2^m)</td>
</tr>
</tbody>
</table>

Table 1: Notation

The code is shown in the order in which it is defined in the original source, except that the definition of the main copy function is at the beginning instead of the very end.

Though knowledge of SML may be helpful, it should not be required for understanding this paper. The SML used in this program includes the Fox extensions to SML/NJ, specifically the \texttt{Byte1}, \texttt{Byte2}, and \texttt{Byte4} structures that provide operations on 8-bit, 16-bit, and 32-bit unsigned integers; those used in this code are array subscripting and update, bit shifts, logical \texttt{and}, and logical \texttt{or}.

3 Functor Header

```sml
functor Copy (structure TimingBoard: TIMINGBOARD): COPY =
  struct
    (* raised if the copy would access data outside one of the arrays *)
    exception Illegal_Copy of {source_length: int, source_offset: int,
                              bytes: int, dest_length: int, dest_offset: int}
    exception Illegal.Align_Value_In_Copy
    (* never raised; avoids compiler warnings *)
    val copy_counter = TimingBoard.add "copy"
  end
```

4 Function copy

The following assumptions are used in the proof; the proof is only valid if the assumptions hold.
1. The computations cause no arithmetic overflow.
2. **TimingBoard.time** returns the result of applying its function to unit, so we can ignore **TimingBoard.time** when verifying correctness.
3. Any out-of-bounds array access will raise the exception **Subscript**.
4. \( \text{Byte4.endian} = \text{Byte4.Little} \) implies that \( \text{Byte2.update}(a, 0, x); \text{Byte2.update}(a, 2, y) \simeq \text{Byte4.update}(a, 0, x | y \ll 16) \)
5. \( \text{Byte4.endian} \neq \text{Byte4.Little} \) implies that \( \text{Byte2.update}(a, 0, x); \text{Byte2.update}(a, 2, y) \simeq \text{Byte4.update}(a, 0, x \ll 16 | y) \)

### 4.1 Preconditions
1. \( \text{src} \neq \text{dest} \) (if \( \text{src} = \text{dest} \), the meaning of the function is undefined).
2. \( \forall j \in \{0 \ldots \text{length}(\text{dest}) - 1\} : \text{dest}_j = \text{orig}_j\).

### 4.2 Specification
1. \( \forall i \in \{0 \ldots \text{length}(\text{src}) - 1\} : \text{the value of src}_i \text{ is not changed by this call.} \)
2. if \( \text{bytes} \leq 0 \) then \( \text{copy} \simeq \text{null function. After the call, } \forall j \in \{0 \ldots \text{length}(\text{dest}) - 1\} : \text{dest}_j = \text{orig}_j \).
3. otherwise if \( \text{srcpos} < 0 \lor \text{destpos} < 0 \) then \( \text{copy} \simeq \text{raise Illegal.Copy} \).
4. otherwise if \( \text{srcpos} + \text{bytes} - 1 \geq \text{length}(\text{src}) \lor \text{destpos} + \text{bytes} - 1 \geq \text{length}(\text{dest}) \) then \( \text{copy} \simeq \text{raise Illegal.Copy} \).
5. otherwise, after the call \( \forall j \in \{0 \ldots \text{length}(\text{dest}) - 1\} : \text{if } \text{destpos} \leq j < \text{destpos} + \text{bytes} \text{ then } \text{dest}_j = \text{src}_{j + \text{srcpos} - \text{destpos}}, \text{otherwise } \text{dest}_j = \text{orig}_j \)

Note that when **Illegal.Copy** is raised, \( \text{dest} \) may or may not be modified.

### 4.3 Proof
The specification of **copy** can be mapped to that of **unaligned** by the following changes:

1. use \( \text{endsrc} \) in **unaligned** instead of \( \text{srcpos} + \text{bytes} \) in **copy**.
2. **unaligned** raises **Subscript** exactly when **copy** raises **Illegal.Copy**.

With these changes, **copy** satisfies its specification if it is equivalent to calling **unaligned**.

**copy** has no side effects except those produced by calling **common**, **aligned**, **semialigned**, or **unaligned**; in every case **copy** will call exactly one of these functions, with as \( \text{endsrc} \) argument the value of \( \text{srcpos} + \text{bytes} \). Since all these functions satisfy the specification of **unaligned** if their preconditions are met, **copy** satisfies its specification if the preconditions of each function are met.

The preconditions of **unaligned** and **common** are the same as those of **copy**.

The preconditions of **aligned** are that \( \text{srcpos} \mod 4 = \text{destpos} \mod 4 = \text{srcalign} \) and \( \text{bytes} \geq 4 \), and these are guaranteed by the negation of \( \text{bytes} < 25 \) and the assertion of \( \text{srcalign} = \text{destalign} \).
The preconditions of semialigned are that srcpos mod 4 = (destpos + 2) mod 4 = srcalign and bytes ≥ 12, and these are guaranteed by the negation of bytes < 25 and of srcalign = destalign, which means that srcpos mod 4 ≠ destpos mod 4, and the assertion of Bits.andb (srcalign + destalign, 0x1) = 0, which means that srcpos mod 2 = destpos mod 2, so that we can conclude that srcpos mod 4 = destpos + 2 mod 4.

The copy function therefore satisfies the specification.

4.4 Code

fun copy (src, srcpos, bytes, dest, destpos) =
    (TimingBoard.time (copy_counter, fn () =>
    (if bytes < 25 then
        (* cannot call aligned or semialigned with less than 12 bytes,
        but we go up to 25 to optimize common cases. *)
        common (src, srcpos, srcpos + bytes, dest, destpos)
    else
        let val srcalign = Bits.andb (srcpos, 0x3)
            val destalign = Bits.andb (destpos, 0x3)
            val endsrc = srcpos + bytes
        in if srcalign = destalign then
            aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes)
        else if Bits.andb (srcalign + destalign, 0x1) = 0 then
            semialigned (src, srcpos, endsrc, dest, destpos, srcalign, bytes)
        else
            unaligned (src, srcpos, endsrc, dest, destpos)
        end)
    handle Subscript => illegal (src, srcpos, bytes, dest, destpos))
    end (* struct *)

5 Function illegal

5.1 Specification

illegal ≡ raise Illegal.Copy

5.2 Proof

If we ignore the print statement, clearly illegal ≡ raise Illegal.Copy. However, the print statement terminates and modifies no existing memory, so it does not change the semantics of illegal.
5.3 Code

\begin{verbatim}
fun illegal (src, srcpos, bytes, dest, destpos) =
  (print ("copy.fun: illegal copy (
    makestring (ByteArray.length src) "", "
    makestring srcpos "", "
    makestring bytes "", "
    makestring (ByteArray.length dest) "", "
    makestring destpos "");
  raise Illegal_Copy {source.length = ByteArray.length src,
    source_offset = srcpos,
    bytes = bytes,
    dest_length = ByteArray.length dest,
    dest_offset = destpos});
\end{verbatim}

6 Function unaligned

This is the basic function for copying bytes. We prove that this is correct, then transform it to the more specific, more optimized versions.

6.1 Preconditions

1. src \neq dest (if src = dest, the meaning of the function is undefined).

2. \forall j \in \{0...\text{length(dest)} - 1\} : \text{dest}_j = \text{orig}_j.

6.2 Specification

1. \forall i \in \{0...\text{length(src)} - 1\} : the value of src\_i is not changed by this call.

2. if \text{endsrc} - \text{srcpos} \leq 0 then \text{copy} \simeq \text{null function. After the call, } \forall j \in \{0...\text{length(dest)} - 1\} : \text{dest}_j = \text{orig}_j.

3. otherwise if \text{srcpos} < 0 \lor \text{destpos} < 0 then \text{copy} \simeq \text{raise Subscript.}

4. otherwise if \text{srcpos} + \text{endsrc} - \text{srcpos} - 1 \geq \text{length(src)} \lor \text{destpos} + \text{endsrc} - \text{srcpos} - 1 \geq \text{length(dest)} then \text{copy} \simeq \text{raise Subscript.}

5. otherwise, after the call \forall j \in \{0...\text{length(dest)} - 1\} : if \text{destpos} \leq j < \text{endsrc} + \text{destpos} - \text{srcpos} then \text{dest}_j = \text{src}_{j+\text{srcpos}+\text{destpos}}, otherwise \text{dest}_j = \text{orig}_j.

6.3 Proof

The sections of this proof correspond to those of the specification.

1. There is only one update operation in the code, and it updates dest; therefore, src is not modified.

2. if \text{endsrc} - \text{srcpos} \leq 0 then \text{srcpos} \geq \text{endsrc}. On the first call to \text{loop}, i = \text{srcpos}, so \text{i} \geq \text{endsrc}, the else clause is not entered, and the function terminates immediately. Therefore, \text{unaligned} \simeq \text{null function.
3. otherwise, if \( \text{srcpos} < 0 \) or \( \text{destpos} < 0 \) then on the first call to \( \text{loop} \), when \( i = \text{srcpos} \) and \( j = \text{destpos} \) and since \( i < \text{endsrc} \), we read \( \text{src}_i \) and update \( \text{dest}_j \). Therefore if \( \text{srcpos} < 0 \) we raise \( \text{Subscript} \), and likewise if \( \text{destpos} < 0 \).

4. otherwise, if \( \text{srcpos} + \text{endsrc} - \text{srcpos} - 1 \geq \text{length}(\text{src}) \) then \( \text{endsrc} - 1 \geq \text{length}(\text{src}) \). Since \( \text{srcpos} < \text{endsrc} \), the else part of the loop is entered at least once when \( i = \text{endsrc} - 1 \), and the operation \( \text{Byte4.sub} (\text{src}, i) \) will then raise \( \text{Subscript} \). Replacing \( \text{srcpos} \) by \( \text{destpos} \), \( i \) by \( j \), and \( \text{src} \) by \( \text{dest} \) proves that if \( \text{destpos} + \text{endsrc} - \text{srcpos} - 1 \geq \text{length}(\text{dest}) \) then \( \text{unaligned} \equiv \text{raiseSubscript} \).

5. otherwise, we have that \( \text{srcpos} \geq 0 \), \( \text{destpos} \geq 0 \), \( \text{srcpos} < \text{endsrc} \leq \text{length}(\text{src}) \), and \( \text{destpos} < \text{destpos} + \text{endsrc} - \text{srcpos} \leq \text{length}(\text{dest}) \). Therefore, the else part of the loop is entered \( \text{endspos} - \text{srcpos} - 1 \) times, with \( i \) taking on all the values in the range \( \text{srcpos} \ldots \text{endspos} - 1 \) inclusive and \( j \) the values in the range \( \text{destpos} \ldots \text{destpos} + \text{endspos} - \text{srcpos} - 1 \) inclusive; \( i \) and \( j \) are related by \( i = j + \text{srcpos} - \text{destpos} \).

Therefore \( \text{unaligned} \) terminates, and when it terminates, \( \forall j \in \{\text{destpos} \ldots \text{destpos} + \text{endsrc} - \text{srcpos} - 1\} : \text{dest}_j = \text{src}_j + \text{srcpos} - \text{destpos} \). The other locations in \( \text{dest} \) are unchanged: \( \forall j \in \{0 \ldots \text{length}(\text{dest}) - 1\} - \{\text{destpos} \ldots \text{destpos} + \text{endsrc} - \text{srcpos} - 1\} : \text{dest}_j = \text{orig}_j \).

6.4 Code

\[
\text{fun unaligned} (\text{src}, \text{srcpos}, \text{endsrc}, \text{dest}, \text{destpos}) = \\
\quad \text{let fun loop} (i, j) = \\
\quad \quad \text{if} \ i \geq \text{endsrc} \ \text{then} \ () \ \\
\quad \quad \text{else} \\
\quad \quad \quad \text{Byte4.update} (\text{dest}, j, \text{Byte4.sub} (\text{src}, i)); \\
\quad \quad \quad \text{loop} (i + 1, j + 1)) \\
\quad \text{in loop} (\text{srcpos}, \text{destpos}) \\
\end{align*}

7 Function common

This function optimizes the implementation of short copies of 1, 2, 3, 4, 8, 16, or 20 bytes.

The preconditions and specification are the same as for \( \text{unaligned} \).

7.1 Proof

We can transform \( \text{unaligned} \) into this function by a sequence of steps, none of which changes the meaning of the function:

1. Rename \( i \) and \( j \) to \( \text{srcpos} \) and \( \text{destpos} \), and inline \( \text{loop} \).
2. Break into cases, use a call to \( \text{unaligned} \) for the default case, and unroll the loop the appropriate number of times for the specific cases.

7.2 Code

\[
\text{fun common} (\text{src}, \text{srcpos}, \text{endsrc}, \text{dest}, \text{destpos}) = \\
\quad \text{case endsrc - srcpos of} \\
\quad \quad 0 \Rightarrow ()
\]
8 Function sixteen

This function copies bytes when both the source and destination are aligned on four-byte boundaries.

8.1 Preconditions

1. $src \neq dest$ (same as for unaligned).
2. $\forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j$ (same as for unaligned).
3. $srcpos \mod 4 = destpos \mod 4 = 0$
4. $(endsrc - srcpos) \mod 16 = 0$
8.2 Specification

Same as for unaligned.

8.3 Proof

We can transform unaligned into this function through the following steps. None of the steps changes the meaning of the function.

1. Unroll the loop in unaligned four times, to give

   ```
   fun loop (i, j) = 
   if i >= endsrc then ()
   else
     (Byte1.update (dest, j + 0, Byte1.sub (src, i + 0));
      Byte1.update (dest, j + 1, Byte1.sub (src, i + 1));
      Byte1.update (dest, j + 2, Byte1.sub (src, i + 2));
      Byte1.update (dest, j + 3, Byte1.sub (src, i + 3));
      loop (i + 4, j + 4))
   ```

   The unrolled loop is strongly isomorphic to the original, since \((endsrc - srcpos)\) mod 16 = 0.
   so \((endsrc - srcpos)\) mod 4 = 0.

2. Merge the four calls to Byte1.sub and the four calls to Byte1.update into single calls to Byte4.sub and Byte4.update respectively.

   ```
   fun loop (i, j) = 
   if i >= endsrc then ()
   else
     (Byte4.update (dest, j, Byte4.sub (src, i));
      loop (i + 4, j + 4))
   ```

   Since srcpos mod 4 = destpos mod 4 = 0, this substitution preserves the meaning of the previous loop.

3. Unroll the loop again four times to give the final code. The unrolled loop is strongly isomorphic to the original since \((endsrc - srcpos)\) mod 16 = 0.

8.4 Code

```
local
  fun sixteen (src, srcpos, endsrc, dest, destpos) =
    let fun loop (i, j) =
      if i >= endsrc then ()
      else
        (Byte4.update (dest, j, Byte4.sub (src, i));
         Byte4.update (dest, j + 4, Byte4.sub (src, i + 4));
         Byte4.update (dest, j + 8, Byte4.sub (src, i + 8));
         Byte4.update (dest, j + 12, Byte4.sub (src, i + 12));
         loop (i + 16, j + 16))
    in loop (srcpos, destpos)
end
```
9 Function aligned

9.1 Preconditions
1. src \neq dest (same as for unaligned).
2. \forall j \in \{0, \ldots, \text{length}(dest) - 1\} : dest_j = orig_j (same as for unaligned).
3. srcpos \mod 4 = destpos \mod 4 = srcalign
4. srcpos + bytes = endsrc
5. bytes \geq 4

9.2 Specification
Same as for unaligned.

9.3 Proof
We can transform unaligned into this function through the following steps. None of the steps
changes the meaning of the function.

1. Add the parameters srcalign and bytes and the let:

   fun aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
   let val front = case srcalign of
     0 => 0
     1 => 3
     2 => 2
     3 => 1
   ...
   val backdest = middest + middle
   in unaligned (src, srcpos, endsrc, dest, destpos)
   end (* let *)

   This is equivalent to unaligned since the let only introduces pure, finite computations.
   Since 0 \leq srcalign = srcpos \mod 4 < 4, the exception Illegal Açign.Value_In_Copy is
   never raised.

2. Split the call to unaligned into:

   unaligned (src, srcpos, midsrc, dest, destpos);
   sixteen (src, midsrc, backsrc, dest, middest);
   unalignel (src, backsrc, endsrc, dest, backdest)

   This is equivalent to the original call, since the range srcpos...midsrc...backsrc...endsrc
   is the same as the range srcpos...endsrc, and likewise for dest, provided the preconditions
   of sixteen are maintained:
(a) midsrc mod 4 = 0 follows from precondition 3 and from midsrc mod 4 = (srcpos + (4 - srcalign)) mod 4 = (srcpos + (4 - srcpos mod 4)) mod 4 = (4 + srcpos - srcpos) mod 4 = 0. Likewise middest mod 4 = 0, since middest = destpos + front as midsrc = srcpos + front and srcpos mod 4 = destpos mod 4.

(b) (backs - midsrc) mod 16 = 0 follows from backs - midsrc = middle = rest - (rest mod 16) and rest ≥ 0 since rest = bytes - x for some x < 4 and bytes ≥ 4.

9.4 Code

in (* local *)
  fun aligned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
    let val front = case srcalign of
      0 => 0
      | 1 => 3
      | 2 => 2
      | 3 => 1
      | _ => raise Illegal_A#ign_Value_In_Copy
    val rest = bytes - front
    val tail = Bits.andb (rest, Oxf)
    val middle = rest - tail
    val midsrc = srcpos + front
    val middest = destpos + front
    val backsrc = midsrc + middle
    val backdest = middest + middle
    in unaligned (src, srcpos, midsrc, dest, destpos);
       sixteen (src, midsrc, backsrc, dest, middest);
       unaligned (src, backsrc, endsrc, dest, backdest)
    end (* let *)
  end (* local *)

10 Function eightlittle

10.1 Preconditions

1. src ≠ dest (same as for unaligned).

2. ∀j ∈ {0...length(dest) – 1}: destj = origj (same as for unaligned).

3. srcpos mod 4 = (destpos + 2) mod 4 = 2

4. endsrc > srcpos

5. (endsrc – srcpos) mod 8 = 2

6. Byte2.update(a, 0, x); Byte2.update(a, 2, y) ≈ Byte4.update(a, 0, (x|y ≪ 16))

10.2 Specification

Same as for unaligned.
10.3 Proof

We can transform unaligned into this function through the following steps. None of the steps changes the meaning of the function.

1. Unroll the loop eight times, and since \((endsrc - srcpos) \mod 8 = 2\), at the beginning copy the two bytes that cannot be copied by the loop. Then merge each pair of Byte1.update and Byte1.update operations into the corresponding Byte2 operations. We can do this since \(srcpos \mod 2 = i \mod 2 = destpos \mod 2 = j \mod 2 = 0\).

   let fun loop (i, j) =
   if i >= endsrc then ()
   else (Byte2.update (dest, j + 0, Byte2.sub (src, i + 0));
       Byte2.update (dest, j + 2, Byte2.sub (src, i + 2));
       Byte2.update (dest, j + 4, Byte2.sub (src, i + 4));
       Byte2.update (dest, j + 6, Byte2.sub (src, i + 6));
       loop (i + 8, j + 8))
   in Byte2.update (dest, destpos, Byte2.sub (src, srcpos));
   loop (srcpos + 2, destpos + 2)
   end

2. Introduce a carry, and within the body of the loop add two to \(i\), so now \(j = i + destpos - srcpos - 2\). To keep equivalence with unaligned, read the first two bytes before the loop and assign the final carry at the end of the loop. In this loop, \(i \mod 4 = (srcpos + 2) \mod 4 = 0\) and \(j \mod 4 = destpos \mod 4 = 0\).

   let fun loop (i, j, carry) =
   if i >= endsrc then Byte2.update (dest, j, carry)
   else (Byte2.update (dest, j + 0, carry);
       Byte2.update (dest, j + 2, Byte2.sub (src, i + 0));
       Byte2.update (dest, j + 4, Byte2.sub (src, i + 2));
       Byte2.update (dest, j + 6, Byte2.sub (src, i + 4));
       loop (i + 8, j + 8, Byte2.sub (src, i + 6))
   in loop (srcpos + 2, destpos, Byte2.sub (src, srcpos))

3. Replace the first two Byte2.update and Byte2.sub operations by corresponding Byte4 operations, since \(i \mod 4 = j \mod 4 = 0\). Use precondition 6 to compute the value to be stored.
val makebyte2 = Byte2.from_int o Byte4.to_int
val makebyte4 = Byte4.from_int o Byte2.to_int
fun mergel (low, high) = Byte4.|| (low, Byte4.<< (high, 16))

... else
  let val srcv = Byte4.sub (src, i)
  in Byte4.update (dest, j, mergel (carry, srcv));
  let val carry = makebyte2 (Byte4.>> (srcv, 16))
  in Byte2.update (dest, j + 4, carry);
     Byte2.update (dest, j + 6, Byte2.sub (src, i));
     loop (i + 8, j + 8,
           makebyte4 (Byte2.sub (src, i + 6)))
  end
end
...

4. Replace the remaining Byte2 operations by the corresponding Byte4 operations.

let val srcv = Byte4.sub (src, i)
  in Byte4.update (dest, j, mergel (carry, srcv));
  let val carry = Byte4.>> (srcv, 16)
    val srcv = Byte4.sub (src, i + 4)
  in Byte4.update (dest, j + 4, mergel (carry, srcv));
  loop (i + 8, j + 8,
     Byte4.>> (srcv, 16))
end
end

5. In-line the calls to mergel and compute new i, j as i + 4, j + 4.

... let val srcv = Byte4.sub (src, i)
  in Byte4.update (dest, j,

Byte4.|| (carry,
Byte4.<< (srcv, 16)));

  let val i = i + 4
    val j = j + 4
    val carry = Byte4.>> (srcv, 16)
    val srcv = Byte4.sub (src, i)
  in Byte4.update (dest, j,

Byte4.|| (carry,
Byte4.<< (srcv, 16)));

loop (i + 4, j + 4,
     Byte4.>> (srcv, 16))
...

This gives us the final code for eightlittle.
10.4 Code

local
val makebyte2 = Byte2.from_int o Byte4.to_int
val makebyte4 = Byte4.from_int o Byte2.to_int
fun eightlittle (src, srcpos, endsrc, dest, destpos) = 
  let fun loop (i, j, carry) = 
    if i >= endsrc then Byte2.update (dest, j, makebyte2 carry) 
    else 
      let val srcv = Byte4.sub (src, i) 
      in Byte4.update (dest, j, Byte4.II (carry, Byte4.<< (srcv, 16))); 
      let val i = i + 4 
      val j = j + 4 
      val carry = Byte4.>> (srcv, 16) 
      val srcv = Byte4.sub (src, i) 
      in Byte4.update (dest, j, Byte4.II (carry, Byte4.<< (srcv, 16))); 
      loop (i + 4, j + 4, Byte4.>> (srcv, 16)) 
    end 
  end 
  in loop (srcpos + 2, destpos, makebyte4 (Byte2.sub (src, srcpos))) 
end

11 Function eightbig

11.1 Preconditions

1. \( src \neq dest \) (same as for unaligned).
2. \( \forall j \in \{0 \ldots length(dest) - 1\} : dest_j = orig_j \) (same as for unaligned).
3. \( srcpos \mod 4 = (destpos + 2) \mod 4 = 2 \)
4. \( endsrc > srcpos \)
5. \( (endsrc - srcpos) \mod 8 = 2 \)
6. \( \text{Byte2.update}(a, 0, x); \text{Byte2.update}(a, 2, y) \simeq \text{Byte4.update}(a, 0, (x \ll 16|y)) \)

11.2 Specification

Same as for unaligned.

11.3 Proof

The preconditions are the same as for eightlittle, except for precondition 6 which is reversed. This reversal is expressed in the code by the interchange of \( < \) and \( > \) and by the initial and final
shift operations that handle storing the two bytes of carry in the most significant half of the carry variable.

11.4 Code

```haskell
fun eightbig (src, srcpos, endsrc, dest, destpos) =
    let fun loop (i, j, carry) =
        if i >= endsrc then
            Byte2.update (dest, j, makebyte2 (Byte4.>> (carry, 16)))
        else
            let val srcv = Byte4.sub (src, i)
                in Byte4.update (dest, j,
                    Byte4.|| (carry, Byte4.>> (srcv, 16)));
                let val i = i + 4
                    val j = j + 4
                    val carry = Byte4.<< (srcv, 16)
                    val srcv = Byte4.sub (src, i)
                    in Byte4.update (dest, j,
                        Byte4.|| (carry,
                            Byte4.>> (srcv, 16)));
                        loop (i + 4, j + 4, Byte4.<< (srcv, 16))
                end
            end
        end
    in loop (srcpos + 2, destpos,
            Byte4.<< (makebyte4 (Byte2.sub (src, srcpos)), 16))
    end
```

12 Function eight

12.1 Preconditions

1. src ≠ dest (same as for unaligned).

2. ∀j ∈ {0…length(dest) − 1}: destj = origj (same as for unaligned).

3. srcpos mod 4 = (destpos + 2) mod 4 = 2

4. endsrc > srcpos

5. (endsrc − srcpos) mod 8 = 2

12.2 Specification

Same as for unaligned.

12.3 Proof

The function eight is eightlittle if Byte4.endian = Byte4.little, and eightbig otherwise. Both functions implement the full specification, but they have different preconditions.
If \( \text{Byte4.endian} = \text{Byte4.little} \), Assumption 4 in Section 4 applies and precondition 6 of \( \text{eightlittle} \) is satisfied, so we use \( \text{eightlittle} \).

Otherwise, Assumption 5 in Section 4 applies and precondition 6 of \( \text{eightbig} \) is satisfied, so we use \( \text{eightbig} \).

12.4 Code

```ml
val eight = if Byte4.endian = Byte4.Little then eightlittle
else eightbig
```

13 Function \text{semialigned}

13.1 Preconditions

1. \( \text{src} \neq \text{dest} \) (same as for \text{unaligned}).

2. \( \forall j \in \{0...\text{length(dest)} - 1\} : \text{dest}_j = \text{orig}_j \) (same as for \text{unaligned}).

3. \( \text{srcalign} = \text{srcpos} \mod 4 = (\text{destpos} + 2) \mod 4 \)

4. \( \text{endsrc} - \text{srcpos} = \text{bytes} \)

5. \( \text{bytes} \geq 12 \)

13.2 Specification

Same as for \text{unaligned}.

13.3 Proof

We can transform \text{unaligned} into this function through the following steps. None of the steps changes the meaning of the function.

1. Add the parameters \( \text{srcalign} \) and \( \text{bytes} \) and the \text{let}:

   ```ml
   fun \text{semialigned} \ (\text{src}, \text{srcpos}, \text{endsrc}, \text{dest}, \text{destpos}, \text{srcalign}, \text{bytes}) =
   let val \text{front} = \text{case srcalign of}
         0 => 2
       | 2 => 0
       | 1 => 3
       | 3 => 1
     ...
     val \text{backdest} = \text{middest} + \text{middle}
   in \text{unaligned} \ (\text{src}, \text{srcpos}, \text{endsrc}, \text{dest}, \text{destpos})
   end (* \text{let} *)
   ```

   This is equivalent to \text{unaligned} since the \text{let} only introduces pure, finite computations.

2. Split the call to \text{unaligned} into:
unaligned (src, srcpos, midsrc, dest, destpos);
eight (src, midsrc, backsrc, dest, middest);
unaligned (src, backsrc, endsrc, dest, backdest)

This is equivalent to the original call (since the range srcpos...midsrc...backsrc...endsrc is the same as the range srcpos...endsrc, and likewise for dest), provided the preconditions of eight are maintained:

(a) midsrc mod 4 = 2 follows from precondition 3 and from midsrc mod 4 = (srcpos + front) mod 4 = (srcpos + (6 - srcalign)) mod 4 = (srcpos + (6 - srcpos mod 4)) mod 4 = 2. Likewise (middest + 2) mod 4 = 2, since middest = destpos + front as midsrc = srcpos + front and srcpos mod 4 = (destpos + 2) mod 4.

(b) backsrc > midsrc follows from backsrc - midsrc = middle and front + middle + tail = bytes ≥ 12 (by precondition 5), and since front ≤ 3 and tail ≤ 7, middle ≥ 2.

(c) (backsrc - midsrc) mod 8 = 2 follows from backsrc - midsrc = middle = rest - ((rest - 2) mod 8)) and rest ≥ 2 since rest ≥ middle ≥ 2.

13.4 Code

in (* local *)
  fun semialigned (src, srcpos, endsrc, dest, destpos, srcalign, bytes) =
    let val front = case srcalign of
      0 => 2
    | 2 => 0
    | 1 => 1
    | 3 => 3
    | _ => raise Illegal_Align_Value_In_Copy
          val rest = bytes - front
          val tail = Bits.andb (rest - 2, OX7)
          val middle = rest - tail
          val midsrc = srcpos + front
          val middest = destpos + front
          val backsrc = midsrc + middle
          val backdest = middest + middle
          in unaligned (src, srcpos, midsrc, dest, destpos);
          eight (src, midsrc, backsrc, dest, middest);
          unaligned (src, backsrc, endsrc, dest, backdest)
    end (* let *)
end (* local *).

14 Summary

This paper describes a proof of an optimized, imperative function. On the negative side, the proof is informal in not stating all the axioms, not always referring to the axioms, and having neither lemmas no theorems. The entire program is only about 200 lines in length. The performance of the implementation is about as good as can be obtained using arrays and bounds checks, but slower than an equivalent C implementation using pointers. The proof was done entirely by hand. Finally,
the copy function is relatively easy to specify, so developing the formal specification of the intended semantics of the function was not an obstacle.

On the positive side, we note that unlike much of the published results about program proofs, we have actually designed a specification for and proven a substantial function in a running system, including all the I/O, exceptions, and optimizations one finds in a real-world program. In spite of its brevity, the program is somewhat complex. To see this, note that the proof for the function unaligned is a little more than a page in length. The proof of correctness for the common case is even simpler, much of the space being taken up with the verification of correct behavior under exceptional conditions. In contrast the proof of the function semialigned and of its subroutines takes up several pages without much space being devoted to exceptional cases; a comparison of the source code for the routines is sufficient to understand why. In the function eightlittle, for example, the loop carries around an accumulator, shifts half-words around and uses them to build words, and in short we have not just an imperative program, but an imperative program that relies on the size of storage units and on the bit representation of values. Because of this complexity, we find the fact that the proof techniques are suitable for creating maintainable proofs to be a strong validation of the proof techniques we used.

It should be noted that in an environment where code is actively maintained, there is no certainty of correctness of a program, even with a proof, unless the proof is mechanically checked to correspond to and verify the source code. Even though the proof presented here may prove the correctness of this version of the program, there is no guarantee that future modifications of the program will be accompanied by correct changes in the proof. The effort of doing a program proof does however help uncover many implementation errors, so a proof does increase confidence in the correctness of code.

Many of the results of this work apply to many different proofs and programs. In particular, we claim that the transformational program proofs shown in this paper can be more easily maintained than pre-condition, post-condition style proofs or other conventional techniques. We also claim that developing the program and the proof together is the best way to develop simple proofs for clear and effective programs.

15 Future Work

We can speculate as to why we found it necessary to prove this particular function and not others. The motivations for the proof were twofold: the program was being changed frequently, and the program frequently had undetected bugs that were far from apparent on inspection. The frequent changes were due to our recognizing that this function was a bottleneck in our implementation of network protocols and to our attempts to optimize it; the mistakes may in part be due to the fact that, although a simple program, this is an imperative program with loops over arrays and word operations, and in SML as in other languages, loops over arrays and word operations are often error-prone.

Discussion with Matthias Felleisen made it apparent that the array, position pairs of parameters to the copy functions are really a form of a dynamic array where the first position bytes of the conventional, underlying array are never used. Perhaps if SML or other functional languages would provide arrays with the analogue of the "tail" and pattern matching operations provided for lists, the copy code might be simpler, potentially more efficient (the compiler could optimize the "tail"

3The function is sufficiently simple that the explicit proof could be omitted with little loss of confidence in the correctness of the program.
operation to pointer operations, avoiding the indexing required of arrays), and easier to understand and prove correct. Such “shrinkable” dynamic arrays might provide all of the performance benefits of pointers with none of the safety and obscurity problems pointers introduce. The program might even become sufficiently easy to understand that a correctness proof would no longer be required; this would be a desirable result.

16 Acknowledgements

None of this work would have been possible without the fruitful discussions the author had with Matthias Felleisen; the knowledge and inspiration he provided were both essential to the final result. Robert Harper and Peter Lee provided encouragement, motivation, and hope that the compiler would improve. Nevin Heintze was instrumental in causing the author to do a proof in the first place. Other members of the Fox project, including but not limited to Kenneth Cline, Mark Leone, Brian Milnes, Gregory Morrisett, and David Tarditi have all at one time or another provided useful feedback that led to the present form of the copy code.

References


