Validating Object Code: `strcmp`

Marco Benini
Validating Object Code: strcmp

Marco Benini
Dipartimento di Scienze dell’Informazione
Università degli Studi di Milano
via Comelico 39/41 — 20135, Milano, Italy
benini@dsi.unimi.it

Abstract. In this paper we describe the correctness proof for the strcmp algorithm. We prove correctness of the object code generated by the gcc compiler. The proof is performed in Isabelle/HOL, using the Computer Arithmetic Toolkit.

1 Introduction

The goal of this paper is to show, by means of a real correctness proof, our proposal for an approach to formal verification of object code.

Our choice is strcmp, a standard routine in the libc library. We compiled it using gcc obtaining as an intermediate result an assembly version for the MC68000 microprocessor.

This code is optimized by the compiler and it shows almost every feature which makes difficult to verify object code in a formal way, but still its size is manageable.

The relevant features are:
- bitwise operations,
- non structured jumps,
- no abstract data types.

Our choice was driven by the size of the code (big examples are too long to be explained) and by the fact that every problem comes like in a nutshell.

We hope you will keep in mind these advices when reading the technical explanations that follow.

In the first section we describe our representation and specification of strcmp. In the second section we describe the general ideas which lead to the development of the correctness proof. The last section contains remarks about the obtained result. We provide the complete set of source files for Isabelle as an appendix.

2 The program and its representation

In this section we describe the representation and the specification for strcmp. As already stated in the introduction, we work on the object code as shown in Fig. 1.
The informal specification for `strcmp` says *given two non-null strings, it returns 0 if and only if they are equal*. In order to formalize this statement we need to disambiguate all the obscure points. First of all, every string is a pointer, so a non-null string means that the pointer is not equal to 0. The second point regards the word *given*: it means that we know where the pointers (strings) are stored.

```
1 link  a6,#0
2 movei d3,sp0-
3 movei d2,sp0-
4 movel a6@8,a1
5 movel a6@12,a0
6 moveq #0,d3
7 moveq #0,d2
L8:  8 movel a10+,d1
     9 movel a00+,d0
10 tateb d1
11 jeq  L10
12 cmpe d1,d0
13 jeq  L8
14 movel d1,d3
15 movel d0,d2
16 movel d3,d0
17 subl d2,d0
18 jra  L13
L10: 19 andl #0xFF,d0
20 neg1 d0
L13: 21 movel a6@-8,d2
22 movel a6@-4,d3
23 unlink a6
24 rts
```

*Fig. 1. Assembly source for `strcmp`*

To simplify our task, and to distinguish between a correctness proof for the subroutine, and a correctness proof of the calling procedure, we discard part of the code, and precisely the *envelope* which surrounds the real code, and takes care of retrieving the parameters and returning the exit value. In this way, the code we are going to analyze is shown in Fig. 2

At this point it is possible to give a complete specification for this piece of code: given two strings, `argA` and `argB`, stored into registers `a0` and `a1`, respectively, both well-formed (i.e., non-null and zero-terminated), and the program counter at location 6 in the program, the program will terminate at a time `t` and, at that time, there is a value `x` such that the register `d0` contains zero if and only if the byte pointed by `argA + x` is zero and the byte pointed by `argB + x`
is zero, and, moreover, up to $x$, the strings are equal and every byte pointed by any value between argB and argB + $x$ is non-zero.

Introducing the following abbreviations (where $\zeta$ is the function which calculates the length of a string)

$$
\alpha \ x = \text{ReadByte}_0 (\text{argA} + x) = \text{ReadByte}_0 (\text{argB} + x)
$$

$$
\beta \ x = (\forall y. 0 \leq y < x \rightarrow \alpha \ y \land \text{ReadByte}_0 (\text{argB} + y) \neq 0)
$$

$$
\phi \ x = (\text{ReadByte}_0 (x + \zeta x) = 0 \land
(\forall y. 0 \leq y < \zeta x \rightarrow \text{ReadByte}_0 (x + y) \neq 0))
$$

the precondition becomes

$$
a_0 \ 0 = \text{argA} \land \ a_1 \ 0 = \text{argB} \land \phi \ \text{argA} \land \phi \ \text{argB} \land \text{pc} \ 0 = 6
$$

and the postcondition is

$$
\exists \ x. \beta \ x \land
(d_0 \ t = 0) = (\text{ReadByte}_0 (\text{argA} + x) = 0 \land \text{ReadByte}_0 (\text{argB} + x) = 0).
$$

In order to represent the program and to divide the correctness proof into a suitable number of lemmas, we partition the code into functional units. Every block is a sequential set of instructions which performs an elementary step in the computation. The result of this approach is shown in Fig. 3.

To represent the program means to provide a logical formula which encodes the computation a single block performs. The general shape of a representation is

$$
\forall t. \text{pc} \ t = B \rightarrow \text{pc} \ (t + \Delta) = E \land \ldots
$$

where $B$ is the location of the first instruction, $E$ is the location where the program will be after executing the block, $\Delta$ is the amount of time (number of
6 moveq  #0,d3
7 moveq  #0,d2

L8:  8 moveb  a18+,d1
     9 moveb  a08+,d0
    10 tstb   d1
    11 jeq    L10

12 cmpb  d1,d0
13 jeq    L8

14 moveb  d1,d3
15 moveb  d0,d2
16 movel  d3,d0
17 subl   d2,d0
18 jra    L13

L10: 19 andl  #0xFF,d0
     20 negl  d0

L13:  21

**Fig. 3.** Dividing the source code into blocks

instructions) the code takes to execute, and the ellipsis stand for the final values of registers. In order to keep our representations short (that means, manageable), we represent just the relevant modifications to registers. The result of this representational effort is shown in Fig. 4.

Since every block is just a piece of code, we want to provide pre- and postconditions for every block, so to divide the complexity of the correctness proof among many lemmas. Essentially this amounts to move specification through blocks. Since we have a precondition and a postcondition for the whole program, we have two starting points for this process. By inspecting and analyzing the code we are able to reconstruct a reasonable amount of specifications for every block. Except for the loop between instruction 8 to 13, that we will discuss in the next section, the complete picture is shown in Fig. 5. It contains every information we need in order to start the correctness proof.

### 3 Constructing the proof

As introduced in the previous section, the reason why we divide the program into blocks is to simplify the correctness proof, by introducing lemmas. Every such a lemma has the shape

\[ \text{Rep}_n, \text{Pre}_n \vdash \text{Post}_n \]

where \( \text{Rep}_n \) is the representation of block number \( n \) and \( \text{Pre}_n, \text{Post}_n \) are its pre and postcondition, respectively.

We can roughly divide these lemmas into three categories:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 6 moveq #0,d3 | $\forall t. pc\ t = 6 \rightarrow$
| 7 moveq #0,d2 | $pc\ (t + 2) = 8 \land$
|   | $d_2\ (t + 2) = 0 \land$
|   | $d_3\ (t + 2) = 0 \land$
|   | $a_0\ (t + 2) = a_0\ t \land$
|   | $a_1\ (t + 2) = a_1\ t$
|   |   |
| L8: 8 moveb a10+,d1 | $\forall t. pc\ t = 8 \rightarrow$
| 9 moveb a00+,d0 | $(\text{ReadByte}_0\ (a_1\ t) = 0 \rightarrow pc\ (t + 4) = 19) \land$
| 10 tstb d1 | $(\text{ReadByte}_0\ (a_1\ t) \neq 0 \rightarrow pc\ (t + 4) = 12) \land$
| 11 jeq L10 | $d_0\ (t + 4) = \text{ReadByte}_0\ (a_0\ t) \land$
|   | $d_1\ (t + 4) = \text{ReadByte}_0\ (a_1\ t) \land$
|   | $d_2\ (t + 4) = d_2\ t \land$
|   | $d_3\ (t + 4) = d_3\ t \land$
|   | $a_0\ (t + 4) = a_0\ t + 1 \land$
|   | $a_1\ (t + 4) = a_1\ t + 1$
|   |   |
| 12 cmpb d1,d0 | $\forall t. pc\ t = 12 \rightarrow$
| 13 jeq L8 | $(d_1\ t = d_0\ t \rightarrow pc\ (t + 2) = 8) \land$
|   | $(d_1\ t \neq d_0\ t \rightarrow pc\ (t + 2) = 14) \land$
|   | $d_0\ (t + 2) = d_0\ t \land$
|   | $d_1\ (t + 2) = d_1\ t \land$
|   | $d_2\ (t + 2) = d_2\ t \land$
|   | $d_3\ (t + 2) = d_3\ t \land$
|   | $a_0\ (t + 2) = a_0\ t \land$
|   | $a_1\ (t + 2) = a_1\ t$
|   |   |
| 14 moveb d1,d3 | $\forall t. pc\ t = 14 \rightarrow$
| 15 moveb d0,d2 | $pc\ (t + 5) = 21 \land$
| 16 movel d3,d0 | $d_0\ (t + 5) = d_0\ t - d_1\ t \land$
| 17 subl d2,d0 | $d_1\ (t + 5) = d_1\ t \land$
| 18 jra L13 | $a_0\ (t + 5) = a_0\ t \land$
|   | $a_1\ (t + 5) = a_1\ t$
|   |   |
| L10: 19 andl #0xFF,d0 | $\forall t. pc\ t = 19 \rightarrow$
| 20 negl d0 | $pc\ (t + 2) = 21 \land$
|   | $d_0\ (t + 2) = -(d_0\ t \land 255) \land$
|   | $d_1\ (t + 2) = d_1\ t \land$
|   | $a_0\ (t + 2) = a_0\ t \land$
|   | $a_1\ (t + 2) = a_1\ t$
| L13: 21 |   |

Fig. 4. Logical representation of blocks
\[ a_0 = \text{arg} \land a_1 = \text{arg} \land \phi \land \text{arg} \land \phi \land \text{arg} \land \phi \land \text{arg} \land \phi \land \text{pc} = 0 = 6 \]
\[ a_0 = \text{arg} \land a_1 = \text{arg} \land \phi \land \text{arg} \land \phi \land \text{arg} \land \phi \land \text{arg} \land \phi \land \text{pc} = 0 = 6 \]

| 6 moveq #0, d3 | \( \forall \text{pc} t = 6 \rightarrow \text{pc} (t + 2) = 8 \) \land 
| 7 moveq #0, d2 | \quad d_2 (t + 2) = 0 \land d_3 (t + 2) = 0 \land 
| 8 moveb a10+, d1, d1 | \quad a_0 (t + 2) = a_0 \land a_1 (t + 2) = a_1 t \land 
| 9 moveb a00+, d0, d0 | \forall \text{pc} t = 8 \rightarrow d_0 (t + 4) = \text{ReadByte}_{0} (a_0 t) \land 
| 10 tstb d1, d1 | \quad d_1 (t + 4) = \text{ReadByte}_{0} (a_1 t) \land 
| 11 jeq L10 | \quad d_2 (t + 4) = d_2 t \land d_3 (t + 4) = d_3 t \land 
| 12 cmpb d1, d0, d1 | \quad (\text{ReadByte}_{0} (a_1 t) = 0 \rightarrow \text{pc} (t + 4) = 19) \land 
| 13 jeq L8 | \quad (\text{ReadByte}_{0} (a_1 t) \neq 0 \rightarrow \text{pc} (t + 4) = 12) \land 

\[ \exists t. x. \beta \land (\text{pc} t = 14 \land \text{pc} t = 19) \land a_0 t = \text{arg} + x + 1 \land a_1 t = \text{arg} + x + 1 \land 
| 14 moveb d1, d3, d1 | \quad d_0 t = \text{ReadByte}_{0} (\text{arg} + x) \land d_1 t = \text{ReadByte}_{0} (\text{arg} + x) \land d_2 t = 0 \land 
| 15 moveb d0, d2, d0 | \quad d_3 t = 0 \land a_0 t \neq 0 \land d_0 t \neq a_1 t \\
| 16 movel d3, d0, d1 | \forall \text{pc} t = 12 \rightarrow (d_1 t = d_0 t \rightarrow \text{pc} (t + 2) = 8) \land 
| 17 subl d2, d0, d2 | \quad (d_1 t \neq d_0 t \rightarrow \text{pc} (t + 2) = 14) \land 
| 18 jra L13 | \quad d_0 (t + 2) = d_0 t \land d_1 (t + 2) = d_1 t \land 
| 19 andl #0xFF, d0 | \quad d_2 (t + 2) = d_2 t \land d_3 (t + 2) = d_3 t \land 
| 20 negl d0 | \quad a_0 (t + 2) = a_0 t \land a_1 (t + 2) = a_1 t \land 

\[ \exists t. x. \beta \land d_0 (t + 5) = \text{ReadByte}_{0} (\text{arg} + x) \land 
| 21 moveb d1, d3, d1 | \quad \text{ReadByte}_{0} (\text{arg} + x) \neq 0 \land \text{ReadByte}_{0} (\text{arg} + x) \neq 0 \land \text{pc} (t + 5) = 21 \\
| 22 movel d3, d0, d1 | \exists t. x. \beta \land d_1 t = 0 \land a_0 t = \text{arg} + x + 1 \land a_1 t = \text{arg} + x + 1 \land 
| 23 subl d2, d0, d2 | \quad d_0 t = \text{ReadByte}_{0} (\text{arg} + x) \land d_1 t = \text{ReadByte}_{0} (\text{arg} + x) \land \text{pc} t = 19 \\
| 24 jra L13 | \forall \text{pc} t = 19 \rightarrow (d_1 t = d_0 t \rightarrow \text{pc} (t + 2) = 21) \land 

\[ \exists t. x. \beta \land (d_0 t = \text{ReadByte}_{0} (\text{arg} + x) \land 
| 25 moveb d1, d3, d1 | \quad \text{ReadByte}_{0} (\text{arg} + x) \neq 0 \land \text{ReadByte}_{0} (\text{arg} + x) \neq 0 \land \text{pc} (t + 5) = 21 \\
| 26 movel d3, d0, d1 | \exists t. x. \beta \land d_1 t = 0 \land a_0 t = \text{arg} + x + 1 \land a_1 t = \text{arg} + x + 1 \land 
| 27 subl d2, d0, d2 | \quad d_0 t = \text{ReadByte}_{0} (\text{arg} + x) \land d_1 t = \text{ReadByte}_{0} (\text{arg} + x) \land \text{pc} t = 19 \\
| 28 jra L13 | \quad a_0 (t + 5) = a_0 t \land a_1 (t + 5) = a_1 t \land 

\[ \exists t. x. \beta \land d_0 (t + 2) = -\text{ReadByte}_{0} (\text{arg} + x) \land 
| 29 moveb d1, d3, d1 | \quad \text{ReadByte}_{0} (\text{arg} + x) = 0 \land \text{pc} (t + 2) = 21 \\
| 30 moveb d1, d3, d1 | \exists t. x. \beta \land (d_0 t = \text{ReadByte}_{0} (\text{arg} + x) \land 
| 31 moveb d1, d3, d1 | \quad \text{ReadByte}_{0} (\text{arg} + x) \neq 0 \land \text{ReadByte}_{0} (\text{arg} + x) \neq 0) \land 
| 32 moveb d1, d3, d1 | \quad (d_0 t = -\text{ReadByte}_{0} (\text{arg} + x) \land \text{ReadByte}_{0} (\text{arg} + x) = 0)) \land \text{pc} t = 21 \\
| 33 moveb d1, d3, d1 | \exists t. x. \beta \land (d_0 t = 0) = (\text{ReadByte}_{0} (\text{arg} + x) = 0 \land \text{ReadByte}_{0} (\text{arg} + x) = 0) \\
| 34 moveb d1, d3, d1 | \exists t. x. \beta \land (d_0 t = 0) = (\text{ReadByte}_{0} (\text{arg} + x) = 0 \land \text{ReadByte}_{0} (\text{arg} + x) = 0) \\

\[ \text{Fig. 5. The global picture} \]
- sequential arguments,
- logical arguments,
- loop arguments.

Every category involves a proper set of proving techniques, so the division is meaningful.

A sequential argument is a statement where the precondition is transformed into the postcondition applying the transformation encoded into the block representation. The proving technique in this case is to remove quantifiers (by instantiating the body of the goal) and to simplify the result (sometimes performing arithmetical calculations). In Fig. 6, 7 and 8 are shown the Isabelle proofs for blocks 6-7, 14-18 and 19-20, respectively, which are sequential blocks.

```
val prems = goal ("[| Rep6_7; Pre6_1 |] ==> Post6";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
    (!simpset addsimps [Rep6_7_def, Pre6_def, Post6_def]) 1));
by (eres_inst_tac ["x","#0"] allE 1);
by (extbin_tac 1);
val Proof1 = result();
```

**Fig. 6.** Proof script for block 6-7

```
val prems = goal ("[| Rep14_18; Pre14_1 |] ==> Post14";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
    (!simpset addsimps [Rep14_18_def, Pre14_def,
                         Post14_def]) 1));
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN' asm_full_simp_tac) 1);
val Proof3 = result();
```

**Fig. 7.** Proof script for block 14-18

A logical argument is a statement where the precondition implies the postcondition, not involving any program transformation. There is no general rule on how to prove such a statement, but a good heuristic is to reduce the goal to a propositional formulation and then to use decision procedures. In the proof of `strfunc`, block 21 (Fig. 9) is of this kind.
val prems = goal strmp.thy
"[| Rep19_20; Pre19 [] ==> Post19|];
by (cut_facts_tac prems 1);
by (asm_full_simp_tac
  (!simpset addsimps [Rep19_20_def, Pre19_def, Post19_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "x"] exI 1);
by ((asmnorm_tac THEN'
  (asm_full_simp_tac (!simpset addsimps [ArithLemma1]))) 1);
val Proof4 = result();

Fig. 8. Proof script for block 19-20

val prems = goal strmp.thy
"Pre21 ==> Post21";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Pre21_def, Post21_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "x"] exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac disjE 1);
by (rtac iffI 1);
by (rtac FalseE 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac notE 1);
by (subgoal_tac "?y - ?x = #0 --> ?y = ?x" 1);
by (supinf_tac 2);
by (REPEAT (eresolve_tac [mp,sym] 1));
by ((rotate_tac "1) THEN' Asm_full_simp_tac) 1);
by (REPEAT (eresolve_tac [mp,sym] 1));
by (Asm_full_simp_tac 1);
by (rtac iffI 1);
by (((rotate_tac "1) THEN' Asm_full_simp_tac) 1);
by (subgoal_tac "$" "$" ?x = #0 --> ?z = #0" 1);
by (supinf_tac 2);
by (etac mp 1);
by ((asmnorm_tac THEN'
  (eres_inst_tac ["P", "Q", "$", "Q = #0"] subst)) 1);
by (extbin_tac 1);
by (((rotate_tac "1) THEN' Asm_full_simp_tac) 1);
val Proof5 = result();

Fig. 9. Proof script for block 21
A block which ends with a conditional branch could be regarded, for the proving technique, as the sum of two sequential blocks, one assuming the branching condition, the other assuming its negation, and as a logical block, in the way it combines the postconditions of its sequential counterparts.

In the code of `strcmp` we have a loop (instructions 8 to 13); as usual in object code, the loop is not structured, i.e., it has not one entry point and one exit point, but some conditional branches control the way to exit or to continue the computation.

To reason about a loop involves two different steps:

- proving the correctness of one cycle;
- proving its termination.

The technique to cope with the first goal is not different from a normal branching block. In our case, with the aid of some additional abbreviations, it reduces to Fig. 10.

In order to prove that every unfolding of the cycle is correct, and in order to ensure that no infinite unfolding is possible, we must use induction. The exact form of the induction principle is suggested by the loop structure.

In the case of `strcmp`, we use the Bounded Chain Principle:

$$[p \leq b, P p]$$

$$\vdash$$

$$\exists x. x \leq b \land P x \quad B \lor (\exists y. p < y \leq b \land P y)$$

$$B$$

Premises say that, given a bound $b$ and a set $P$ of integers, there is a point less or equal than $b$ in $P$, and whenever this situation holds, either it is possible to find another point $y$ which replicate the situation, or $B$ holds.

Since integer numbers provide a discrete ordering, sooner or later, we will exhaust the interval $[x, b]$, so, eventually $B$ must hold.

Reading it as a loop, the principle says that, if we can prove that the loop is entered and, for every step, it either exits with postcondition $B$ or it loops again, but decreasing a complexity measure given by the distance of $y$ from $b$, then, eventually, it will exit with postcondition $B$ true.

In our case, $B$ is simply the postcondition of the cycle, as shown in Fig. 5; $P x$ is $\exists t. 0 \leq t \lt x$, i.e., there is a time such that the loop will reach position $x$ in the string argB; and the bound is simply $\zeta$ argB, the length of string argB.

The reason for this bound is simple: if we will scan position $\zeta$ argB, it means that argB is shorter than argA and the loop will exit (see instructions 10-11, in the source code).

Following standard proving techniques for sequential and logical arguments, the correctness of the loop is established (proof script in Fig. 11).

With these lemmas is easy to prove correctness for the whole `strcmp` algorithm, as in Fig. 12; the proof combines lemmas following the flow of control of the program.
\[ \delta x = (a_0 t = \text{arg}_{A} + x + 1 \land a_1 t = \text{arg}_{B} + x + 1 \land \\
\quad d_0 t = \text{ReadByte}_0 (a_0 t + x) \land d_1 t = \text{ReadByte}_0 (a_1 t + x) \land \\
\quad d_2 t = 0 \land d_3 t = 0 \land \beta x \land \\
\quad (p c = 8 \lor p c = 14 \lor p c = 19) \land \\
\quad (p c = 14 \rightarrow \text{ReadByte}_0 (a_0 t + x) \neq 0 \land \\
\quad \text{ReadByte}_0 (a_0 t + x) = \text{ReadByte}_0 (a_1 t + x) \land \\
\quad (p c = 14 \rightarrow \text{ReadByte}_0 (a_1 t + x) \neq 0) \land \\
\quad \text{ReadByte}_0 (a_1 t + x) = \text{ReadByte}_0 (a_1 t + x) \land \\
\quad (p c = 19 \rightarrow \text{ReadByte}_0 (a_2 t + x) = 0)) \]

\[ \gamma x = (\exists t \delta x t) \]

val prems = goal strcmp.thy
  ".waitForRep8_11; Rep12_13; beta x; pc t = #8; #0 <= x; \\
  \quad a0 t = \text{arg}_{A} + x; a1 t = \text{arg}_{B} + x; d2 t = #0; d3 t = #0 ] => \\
  \quad gamma x";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (\{simpset addsimps [Rep8_11_def, Rep12_13_def, \\
  gamma_def, delta_def] \} 1));
by (((eres_inst_tac \{"x","t"\} allE) THEN' \\
  (etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (case_tac "ReadByte (a1 t) #0 = #0" 1);
by (res_inst_tac \{"x","t + #4"\} exI 1);
by (asm_full_simp_tac 1);
by (((eres_inst_tac \{"x","t + #4"\} allE) THEN' \\
  extbin_tac THEN' asmfull_simp_tac) 1);
by (((etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (res_inst_tac \{"x","t + #6"\} exI 1);
by ((asm_full_simp_tac THEN' extbin_tac THEN' asmfull_simp_tac) 1);
by (case_tac "ReadByte (x + argB) #0 = ReadByte (x + argB) #0" 1);
by (ALLGOALS ((rotate_tac "1" THEN' \\
  asmfull_simp_tac THEN'
  (TRY o (etac not_sym))) ));
val Proof2_1 = result () ;

Fig. 10. Proof script which shows that the body of the loop is correct
val prems = goal_strm.p.thy "[| Rep8_11; Rep12_13; Pre8 [] |] ==> Post8"
by (cut_facts_tac prems 1);
by ((asm_full simp_tac (simpset add_simp [Rep8_11_def, Rep12_13_def, 
  Pre8_def, Post8_def, Post6_def]) 1) THEN (asmnorm_tac 1));
by (res_inst_tac ["P", "x. (? t. #0 <= x & delta x t)"]
  ("b", "zeta argB") BCP 1);
by (res_inst_tac ["x", "#0"] exI 1);
by (((REPEAT o (resolve_tac [conjI, zeta_pos, Proof2_2]))
  THEN' supinf_tac) 1);
by (TRYALL (asm_full simp_tac
  (!simpset add_simp [Rep8_11_def, Rep12_13_def, beta_def])));
by (TRYALL supinf_tac);
by (asm_full simp_tac (!simpset add_simp [delta_def]) 1);
by ((asmnorm_tac THEN' (etac exE) THEN' asmnorm_tac) 1);
by (etac disjE 1);
by (((rotate_tac "1" THEN' Asm_full simp_tac) 1);
by (rtac disjI2 1);
by (res_inst_tac ["x", "p + #1"] exI 1);
by (((rtac conjI) THEN' supinf_tac THEN' (rtac conjI)) 1);
by (subgoal_tac "(p <= zeta argB) = (p < zeta argB | p = zeta argB)"
  (atcl 1));
by (Asm_full simp_tac 1);
by (etac disjE 1);
by (supinf one asm_tac "1 1");
by (((rotate_tac "1" THEN'
  (asm_full simp_tac (!simpset add_simp [phi_def])))) 1);
by (supinf_tac 1);
by (((rtac conjI) THEN' (supinf one asm_tac "11") 1);
by (fold_goals_tac [gamma_def, delta_def]);
by (((rotate_tac "11" THEN' (rtac Proof2_1)) 1);
by (TRYALL (asm_full simp_tac
  (!simpset add_simp [Rep8_11_def, Rep12_13_def, beta_def])));
by (TRYALL supinf_tac);
by ((strip_tac THEN' asmnorm_tac) 1);
by (subgoal_tac "(y < p + #1) = (y < p | y = p)"
  (atcl 1));
by (Asm_full simp_tac 1);
by (etac disjE 1);
by (Fast_tac 1);
by (((rotate_tac "1" THEN'
  (asm_full simp_tac (!simpset add_simp [alpha_def])))) 1);
by (supinf_tac 1);
by (supinf one asm_tac 0 1);
by (rtac disjI1 1);
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "p"] exI 1);
by (Asm simp tac 1);
val Proof2 = result();

Fig. 11. Proof script for induction in the loop
val prems = goal strcmp.thy
  "[ | Rep6_7; Rep8_11; Rep12_13; Rep14_18; Rep19_20; Pre |] => Spec"
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Pre_def, Spec_def]) 1);
by (rtac Proof5 1);
by (asm_full_simp_tac (!simpset addsimps [Pre21_def]) 1);
by (subgoal_tac "Pre14 | Pre19" 1);
by (etac disjE 1);
by (((etac (make_elim Proof3)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Post14_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x", "t + #5"] exI 1);
by (res_inst_tac ["x", "z"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof4)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Post19_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x", "t + #2"] exI 1);
by (res_inst_tac ["x", "z"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof2)) THEN' atac) 1);
by (asm_full_simp_tac (!simpset addsimps [Pre8_def]) 1);
by (((etacl Proof1) THEN' atac) 1);
by (asm_full_simp_tac
  (!simpset addsimps [Post8_def, Pre14_def, Pre19_def]) 1);
by ((REPEAT (etacl exE 1)) THEN (asmnorm_tac 1));
by (etacl disjE 1);
by (rtac disjI1 1);
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "z"] exI 1);
by (asm_full_simp_tac 1);
by (rtac disjI2 1);
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "z"] exI 1);
by (asm_full_simp_tac 1);
val strcmp_correctness_proof = result();

Fig. 12. Correctness proof for strcmp
4 Concluding remarks

In this technical report we have shown that the standard library function `strcmp` is correct. We gave a formal proof of this fact using Isabelle/HOL and the Computer Arithmetic Toolkit as a workbench.

The key ideas behind the proof itself are general and it is worth spending few words to summarize them:

- dividing the proof into lemmas suggested by sequential pieces of code makes it possible to reduce complexity of the proof and isolate trivial parts from difficult ones (i.e., sequential and logical arguments from loops).
- to cope with loops we need an induction principle which resembles the structure of the loop itself. Most of the times this reduces to choose a variation of what, in literature, is known as Noetherian Induction.
- the formal verification of object code requires a good treatment of computer arithmetic, since this is the only way a CPU has in order to perform calculations.

In the appendix of this document we include the complete source files for the Isabelle theory where to correctness proof is developed.

We want to point out how much of this effort could be automated:

- the proof of sequential blocks can be automated by developing more general, more flexible, more high-levels tactics.
- the division of code into blocks can be performed automatically and their representation can be computed.
- even proofs of loops could be partially automated by guessing in a mechanical way what is supposed to be the right induction principle.

A Source Files

A.1 `strcmp.thy`

```
strcmp = ExtBin + strcmpdef + ArithSupp +

consts (* Representation of blocks *)
  Rep19_20 :: bool
  Rep14_18 :: bool
  Rep12_13 :: bool
  Rep8_11 :: bool
  Rep6_7 :: bool

defs

Rep19_20_def "Rep19_20 == ! t. pc t = #19 -->
\ pc (t + #2) = #21 &
\ a0 (t + #2) = a0 t &
\ ai (t + #2) = ai t &
\ di (t + #2) = di t &
```

d0 (t + #2) = "$" (d0 t =and #255"

Rep14_18_def "Rep14_18 = ! t. pc t = #14 --> \
\pc (t + #5) = #21 & \
a0 (t + #5) = a0 t & \
a1 (t + #5) = a1 t & \
d1 (t + #5) = d1 t & \
d0 (t + #5) = d0 t - d1 t"

Rep12_13_def "Rep12_13 = ! t. pc t = #12 --> \
\((d1 t = d0 t) --> (pc (t + #2) = #8)) & \n\((d1 t = d0 t) --> (pc (t + #2) = #14)) & \n\a0 (t + #2) = a0 t & \
a1 (t + #2) = a1 t & \
d3 (t + #2) = d3 t & \
d2 (t + #2) = d2 t & \
d0 (t + #2) = d0 t & \
d1 (t + #2) = d1 t"

Rep8_11_def "Rep8_11 = ! t. pc t = #8 --> \
\((\text{ReadByte} (a1 t) #0 = #0) --> \
\((\text{ReadByte} (a1 t) #0 = #0) --> \
\a0 (t + #4) = a0 t + #1 & \
a1 (t + #4) = a1 t + #1"

Rep6_7_def "Rep6_7 = ! t. pc t = #6 --> \
\pc (t + #2) = #8 & \
d3 (t + #2) = #0 & \
d2 (t + #2) = #0 & \
a0 (t + #2) = a0 t & \
a1 (t + #2) = a1 t"

consts  (* Specifications for every block *)
Pre06 :: bool
Post06 :: bool
Pre08 :: bool
Post08 :: bool
Pre14 :: bool
Post14 :: bool
Pre19 :: bool
Post19 :: bool
Pre21 :: bool
Post21 :: bool

defs
Post21_def "Post21 = (? t x. beta x & \
\(d0 t = #0) = \
\"
(ReadByte (argA + x) #0 = #0 & \ 
ReadByte (argB + x) #0 = #0))"

Pre21_def "Pre21 == (? t x. beta x & 
((do t = 
 \ ReadByte (argA + x) #0 - \ 
 \ ReadByte (argB + x) #0 & \ 
 \ ReadByte (argA + x) #0 == \ 
 \ ReadByte (argB + x) #0) | \ 
 \ (do0 t = 
 \ $" ReadByte (argA + x) #0 & \ 
 \ ReadByte (argB + x) #0 = #0 & \ 
 \ pc t = #21))"

Post19_def "Post19 == (? t x. do0 (t + #2) = 
 \ $" ReadByte (argA + x) #0 & \ 
 \ ReadByte (argB + x) #0 = #0 & \ 
 \ beta x & \ 
 \ pc (t + #2) = #21)"

Pre19_def "Pre19 == (? t x. a0 t = argA + x + #1 & 
 \ a1 t = argB + x + #1 & \ 
 \ d0 t = ReadByte (argA + x) #0 & \ 
 \ d1 t = ReadByte (argB + x) #0 & \ 
 \ beta x & \ 
 \ d1 t = #0 & \ 
 \ pc t = #19)"

Post14_def "Post14 == (? t x. do0 (t + #5) = 
 \ ReadByte (argA + x) #0 - \ 
 \ ReadByte (argB + x) #0 & \ 
 \ ReadByte (argA + x) #0 == \ 
 \ ReadByte (argB + x) #0 & \ 
 \ beta x & \ 
 \ pc (t + #5) = #21)"

Pre14_def "Pre14 == (? t x. a0 t = argA + x + #1 & 
 \ a1 t = argB + x + #1 & \ 
 \ d0 t = ReadByte (argA + x) #0 & \ 
 \ d1 t = ReadByte (argB + x) #0 & \ 
 \ beta x & \ 
 \ d2 t = #0 & \ 
 \ d3 t = #0 & \ 
 \ d1 t == #0 & \ 
 \ d0 t == d1 t & \ 
 \ pc t = #14)"

Post8_def "Post8 == (? t x. #0 <= x & 
 \ a0 t = argA + x + #1 & 
 \ a1 t = argB + x + #1 & \ 
 \ d0 t = ReadByte (argA + x) #0 & \ 
 \ d1 t = ReadByte (argB + x) #0 & \ 
 \ (pc t = #14 | pc t = #19) & \ 
 \ beta x & \ 
 \ d2 t = #0 & \ 
 \ d3 t = #0 & \ 

\( \text{(pc t = #14 -> d1 t = #0 &} \\text{)} \)
\( \text{d0 t = d1 t)} \& \text{)} \)
\( \text{(pc t = #19 --> d1 t = #0))} \)

Pre8_def  "Pre8 == Post6"
Post6_def  "Post6 == (phi argA & \`
phi argB & \`
 a0 #2 = argA & \`
da1 #2 = argB & \`
da2 #2 = #0 & \`
da3 #2 = #0 & \`
\text{pc #2 = #8)"
Pre6_def  "Pre6 == (a0 #0 = argA & \`
a1 #0 = argB & \`
\text{phi argA & \`
\text{phi argB & \`
\text{pc #0 = #6)"
}

consts  (* Specification of the whole algorithm *)
Spec :: bool
Pre :: bool
defs
Spec_def  "Spec == Post21"
Pre_def  "Pre == Pre6"
end

A.2 strcmp.ML

(* --------------------------------------------------------- *)
(* Some useful tactics *)

fun supinf_oneasm_tac a n =
(rotate_tac a n) THEN
(rtac mp n) THEN
(atac (n + 1)) THEN
(supinf_tac n);

fun asnorm_tac n = REPEAT (etac conjE n);

(* --------------------------------------------------------- *)
(* Proof 1: correctness of block [6,7] *)

val prems = goal strcmp.thy
"[[ Rep6_7; Pre6 |] => Post6];
b by (cut_facts_tac prems i);
b by (asm_full_simp_tac
  (simpset addsimps [Rep6_7_def, Pre6_def, Post6_def]) i);
b by (eres_inst_tac [("\","#0") allE 1);

by (exthin_tac 1);
val Proof1 = result();

(* --------------------------------------------------------------------- *)
(* Proof 2: correctness of block [8-13] *)

(* Proof 2.1: the induction step in the loop *)

val prems = goal strcmp.thy
  "[| Rep8_i1; Rep12_i13; beta x; \\
  \ pc t = #8; #0 <= x; \\
  \ a0 t = argA + x; a1 t = argB + x; \\
  \ d2 t = #0; d3 t = #0 |] ==> \gamma x'';;
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps
  [Rep8_i1_def, Rep12_i13_def, gamma_def, delta_def]) 1);
by (((eres_inst_tac ["x","t"] allE) THEN'
  (etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (case_tac "ReadByte (a1 t) #0 = #0" 1);
by (res_inst_tac ["x","t + #4"] ex1 1);
by (Asm_full_simp_tac 1);
by (((eres_inst_tac ["x","t + #4"] allE) THEN'
  extbin_tac THEN' Asm_full_simp_tac) 1);
by (((etac (make_elim mp)) THEN' atac THEN' asmnorm_tac) 1);
by (res_inst_tac ["x","t + #6"] ex1 1);
by ((Asm_full_simp_tac THEN' extbin_tac THEN'
  Asm_full_simp_tac) 1);
by (case_tac "ReadByte (x + argB) #0 = ReadByte (x + argA) #0" 1);
by (ALLGOALS (rotate_tac ~1) THEN'
  Asm_full_simp_tac THEN'
  (TRY o (etac not_sym))));
val Proof2_1 = result();

(* Proof 2.2: a simple tautology *)

val prems = goal strcmp.thy
  "(B & (? t. P t)) ==> (? t. (B & P t))";
by (cut_facts_tac prems 1);
by (Fast_tac 1);
val Proof2_2 = result();

(* Proof 2: correctness of the loop *)

val prems = goal strcmp.thy
  "[| Rep8_i1; Rep12_i13; Pre8 [] ==> Post8";
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps
  [Rep8_i1_def, Rep12_i13_def,
   Pre8_def, Post8_def, Post6_def]) 1);
by (asmnorm_tac 1);

  (* the proof is by induction on (? t. #0 <= x & delta x t) *)
  (* base step *)
by (res_inst_tac ["P", "% x. (? t. #0 <= x & delta x t)"],
      ["b", "zeta argB"] BCP 1);
by (res_inst_tac ["x", "#0"] exI 1);
by (((REPEAT o (resolve_tac [conjI, zeta_pos, Proof2_2]))
      THEN' supinf_tac) 1);
by (fold_goals_tac [gamma_def]);
by (rtac Proof2_1 1);
by (TRYALL (asm_full_simp_tac ![simpset addsimps
                      [Rep8_11_def, Rep12_13_def, beta_def])));
by (TRYALL supinf_tac);
  (* inductive step *)
by (asm_full_simp_tac ![simpset addsimps [delta_def]] 1);
by ((asmnorm_tac THEN' (etac exE) THEN' asmnorm_tac) 1);
by (etac disjE 1);
by (((rotate_tac "1" THEN' Asm_full_simp_tac) 1);
by (rtac disjI2 1);
by (res_inst_tac ["x", "p + #1"] exI 1);
by (((rtac conjI) THEN' supinf_tac THEN' (rtac conjI) 1);
by (subgoal_tac
 "(p <= zeta argB) = (p < zeta argB | p = zeta argB)" 1);
by (Asm_full_simp_tac 1);
by (etac disjE 1);
by (((rotate_tac "1" THEN'
      (asm_full_simp_tac ![simpset addsimps [phi_def]]) 1)) 1);
by (supinf_tac 1);
by (((rtac conjI) THEN' (supinf_oneasm_tac "11") 1)) 1);
by (fold_goals_tac [gamma_def, delta_def]);
by (((rotate_tac "11" THEN' (rtac Proof2_1) 1)) 1);
by (TRYALL (asm_full_simp_tac ![simpset addsimps
                      [Rep8_11_def, Rep12_13_def, beta_def]) 1));
by (TRYALL supinf_tac);
by ((strip_tac THEN' asmnorm_tac) 1);
by (subgoal_tac "(y < p + #1) = (y < p | y = p)" 1);
by (Aas_full_simp_tac 1);
by (etac disjE 1);
by (Fast_tac 1);
by (((rotate_tac "1" THEN'
      (asm_full_simp_tac ![simpset addsimps [alpha_def]]) 1)) 1);
by (supinf_tac 1);
by (supinf_oneasm_tac 0 1);
by (rtac disjI1 1);
by (res_inst_tac ["x", "t"] exI 1);
by (res_inst_tac ["x", "p"] exI 1);
by (Asm_simp_tac 1);
val Proof2 = result();
(* ------------------------------------------------------------------------------------------------- *)
(* Proof 3: correctness of block [14--18] *)

val prems = goal stdclos.thy
"[[ Rep14_18; Pre14 ]] ==> Post14"
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Rep14_18_def, Pre14_def, Post14_def]) 1);
by (REPEAT (etaC ex1 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
val Proof3 = result();

(* ------------------------------------------------------------------------------------------------- *)
(* Proof 4: correctness of block [19,20] *)

val prems = goal stdclos.thy
"[[ Rep19_20; Pre19 ]] ==> Post19"
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Rep19_20_def, Pre19_def, Post19_def]) 1);
by (REPEAT (etaC ex1 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1); Val Proof4 = result();

(* ------------------------------------------------------------------------------------------------- *)
(* Proof 5: the exit condition implies the specification *)

val prems = goal stdclos.thy
"Pre21 ==> Post21"
by (cut_facts_tac prems 1);
by (asm_full_simp_tac (!simpset addsimps [Pre21_def, Post21_def]) 1);
by (REPEAT (etaC ex1 1));
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac disjE 1);
by (rtac iffI 1);
by (rtac FalseE 1);
by ((asmnorm_tac THEN' Asm_full_simp_tac) 1);
by (etac notE 1);
by (subgoal_tac "?y - ?x = #0 --> ?y = ?x" 1);
by (supinf_tac 2);
by (REPEAT (eresolve_tac [mp,sym] 1));
by (((rotate_tac "1" THEN' Asm_full_simp_tac) 1));
by (REPEAT (eresolve_tac [mp, sym] 1));
by (Asm_full_simp_tac 1);
by (rtac iffI 1);
by (((rotate_tac "1" THEN' Asm_full_simp_tac) 1));
by (subgoal_tac " $\$ \$ ?x = #0 \rightarrow ?x = #0" 1);
by (supinf_tac 2);
by (etac mp 1);
by ((asmnorm_tac THEN'
  (eres_inst_tac ["\"^\","% Q. $\$ Q = #0\"] subst) 1));
by (extbin_tac 1);
by (((rotate_tac "1" THEN' Asm_full_simp_tac) 1));
val Proof5 = result ();

(* -------------------------------------------------------------- *)
(* Correctness proof for the whole algorithm *)

val prems = goal_str cmp.thy
 "[! Rep0_7; Rep8_11; Rep12_13; \
   Rep14_18; Rep19_20; Pre [](== Spec"
by (cut_facts_tac prems 1);
by (Asm_full_simp_tac (!simpset addsimp [Pre_def, Spec_def]) 1);
by (rtac Proof5 1);
by (Asm_full_simp_tac (!simpset addsimp [Pre21_def]) 1);
by (subgoal_tac "Pre14 | Pre19" 1);
by (etac disjE 1);
by (((etac (make_elim Proof3)) THEN' atac) 1);
by (Asm_full_simp_tac (!simpset addsimp [Post14_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t + #5"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof4)) THEN' atac) 1);
by (Asm_full_simp_tac (!simpset addsimp [Post9_def]) 1);
by (REPEAT (etac exE 1));
by (res_inst_tac ["x","t + #2"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Fast_tac 1);
by (((etac (make_elim Proof2)) THEN' atac) 1);
by (Asm_full_simp_tac (!simpset addsimp [Post8_def]) 1);
by (((etac Proof1) THEN' atac) 1);
by (Asm_full_simp_tac (!simpset addsimp
 [Post8_def, Pre14_def, Pre19_def] 1));
by (REPEAT (etac exE 1)) THEN (asmnorm_tac 1))
by (etac disjE 1);
by (rtac disjI 1);
by (res_inst_tac ["x","t"] exI 1);
by (res_inst_tac ["x","x"] exI 1);
by (Asm_full_simp_tac 1);
by (rtac disjI2 1);
A.3 strcmpdef.thy

\texttt{strcmpdef} = ExtBin +

\texttt{consts}

\texttt{mm :: [ int, int ] => int}

\texttt{sp :: int => int}
\texttt{pc :: int => int}

\texttt{d0 :: int => int}
\texttt{d1 :: int => int}
\texttt{d2 :: int => int}
\texttt{d3 :: int => int}
\texttt{d4 :: int => int}
\texttt{d5 :: int => int}
\texttt{d6 :: int => int}
\texttt{d7 :: int => int}

\texttt{a0 :: int => int}
\texttt{a1 :: int => int}
\texttt{a2 :: int => int}
\texttt{a3 :: int => int}
\texttt{a4 :: int => int}
\texttt{a5 :: int => int}
\texttt{a6 :: int => int}
\texttt{a7 :: int => int}

\texttt{consts}

\texttt{ReadByte :: [ int, int ] => int}
\texttt{ReadWord :: [ int, int ] => int}
\texttt{ReadLong :: [ int, int ] => int}
\texttt{WriteByte :: [ int, int, int ] => bool}
\texttt{WriteWord :: [ int, int, int ] => bool}
\texttt{WriteLong :: [ int, int, int ] => bool}

\texttt{defs}

\texttt{ReadByte_def = ReadByte p t == mm p t} \
\texttt{ReadWord_def = ReadWord p t == \}
\texttt{ \ \ \ \ ReadByte p t + #256 * ReadByte (p + #1) t} \
\texttt{ReadLong_def = ReadLong p t == \}
\texttt{ \ \ \ \ ReadWord p t + #65536 * ReadWord (p + #2) t} \
\texttt{WriteByte_def = WriteByte p t v == \}
\texttt{ \ \ \ \ v < #256 \ \ \ \ & WriteByte p t = v}
WriteWord_def "WriteWord p t v == \\
  v < #65536 & ReadWord p t = v"
WriteLong_def "WriteLong p t v == ReadLong p t = v"

consts
  argA :: int
  argB :: int
  alpha :: int => bool
  beta :: int => bool
  gamma :: int => bool
  delta :: int => int => bool
  phi :: int => bool
  zeta :: int => int

defs
  alpha_def "alpha x == (ReadByte (argA + x) #0 = \n       ReadByte (argB + x) #0)"
  beta_def "beta x == (! y. #0 <= y & y < x --> alpha y & \n       ReadByte (argB + y) #0 "= #0)"
  gamma_def "gamma x == (? t. delta x t)"
  delta_def "delta x t == (a0 t = argA + x + #1 & \n     a1 t = argB + x + #1 & \n     d0 t = ReadByte (argA + x) #0 & \n     d1 t = ReadByte (argB + x) #0 & \n     d2 t = #0 & \n     d3 t = #0 & \n     beta x & \n     (pc t = #8 | \n      pc t = #14 | \n      pc t = #19) & \n     (pc t = #8 --&gt; \n      ReadByte (argB + x) #0 "= #0 & \n      ReadByte (argA + x) #0 = \n      ReadByte (argB + x) #0 & \n     (pc t = #14 --&gt; \n      ReadByte (argB + x) #0 "= #0 & \n      ReadByte (argA + x) #0 = \n      ReadByte (argB + x) #0 & \n     (pc t = #19 --&gt; \n      ReadByte (argB + x) #0 = #0))"
  phi_def "phi z == (ReadByte (z + zeta z) #0 = #0 & \n    (! y. #0 <= y & y < zeta z --&gt; \n     ReadByte (z + y) #0 "= #0))"

rules
  zeta_pos "#0 &lt; zeta x"

  BCP "[! ? x. (x &lt;= b &amp; P x); \n       ! p. ![ p &lt;= b; P p ]] ==&gt; \n     ![ ! x. (x &lt;= b &amp; P x) &amp; ![ ! p. ![ p &lt;= b; P p ]; ! p. ![ p &lt;= b; P p ]] &lt;= b &amp; P x]"
A.4 ArithSupp.thy

ArithSupp = strcmpdef +

rules
  ArithLemmal "ReadByte x y zand #255 = ReadByte x y"

end