Computer Arithmetic
Aliasing issues: Call by reference, Pointer programs

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

24 janvier 2014
Outline

Exercises from last lecture

Computer Arithmetic
   Handling Machine Integers
   Floating-Point Computations

Aliasing Issues
   Call by Reference
   Pointer Programs
Example: McCarthy’s 91 Function

\[ f_{91}(n) = \text{if } n \leq 100 \text{ then } f_{91}(f_{91}(n + 11)) \text{ else } n - 10 \]

Exercise: find adequate specifications.

```ml
let fun f91(n:int): int
    requires ?
    variant ?
    writes ?
    ensures ?
body
    if n ≤ 100 then f91(f91(n + 11)) else n - 10
```

See `f91.mlw`
Implement and prove binary search using a immediate exit:

\[ low = 0; \quad high = n - 1; \]
while \( low \leq high \):
    let \( m \) be the middle of \( low \) and \( high \)
    if \( a[m] = v \) then return \( m \)
    if \( a[m] < v \) then continue search between \( m \) and \( high \)
    if \( a[m] > v \) then continue search between \( low \) and \( m \)

See `bin_search.mlw`
Exercise: Selection Sort

```ml
val a : array real

let rec sort ()
  writes a
  ensures ?
  body ?
```

1. Formalize postconditions:
   - array in increasing order between 0 and \( n - 1 \),
   - array at exit is a permutation of the array at entrance.

2. Implement and prove selection sort algorithm:
   for each \( i \) from 0 to \( n - 1 \):
      find index \( idx \) of the min element between \( i \) and \( n - 1 \)
      swap elements at indexes \( i \) and \( idx \)

See `sel_sort.mlw`
Outline

Exercises from last lecture

Computer Arithmetic
   Handling Machine Integers
   Floating-Point Computations

Aliasing Issues
   Call by Reference
   Pointer Programs
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - \( 2147483647 + 1 \rightarrow -2147483648 \)
  - \( 100000^2 \rightarrow 1410065408 \)
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may **overflow**
  - \(2147483647 + 1\) → \(-2147483648\)
  - \(100000^2\) → 1410065408

- **floating-point numbers** (32-, 64-bit):
  - **overflows**
    - \(2 \times 2 \times \cdots \times 2\) → \(+\text{inf}\)
    - \(-1/0\) → \(-\text{inf}\)
    - \(0/0\) → NaN
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$

- floating-point numbers (32-, 64-bit):
  - overflows
    - $2 \times 2 \times \cdots \times 2 \rightarrow +\text{inf}$
    - $-1/0 \rightarrow -\text{inf}$
    - $0/0 \rightarrow \text{NaN}$
  - rounding errors
    - $0.1 + 0.1 + \cdots + 0.1 = 1.0 \rightarrow \text{false}$
      \[10\text{times}\]
      (because $0.1 \rightarrow 0.100000001490116119384765625$ in 32-bit)

See also arith.c
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
▶ 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
- 2007, Excel displays 77.1 × 850 as 100000.
Some Numerical Failures

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.

Internal clock ticks every \textit{0.1 second}.  
Time is tracked by \textit{fixed-point arith.}: \(0.1 \approx 209715 \cdot 2^{-24}\).  
Cumulated skew after 24h: \(-0.08\)s, distance: 160m.  
System was supposed to be rebooted periodically.

▶ 2007, Excel displays \(77.1 \times 850\) as 100000.  
Bug in \textit{binary/decimal conversion}.  
Failing inputs: 12 FP numbers.  
Probability to uncover them by random testing: \(10^{-18}\).
Integer overflow: example of Binary Search

- Google “Read All About It: Nearly All Binary Searches and Mergesorts are Broken”

```ml
let l = ref 0 in
let u = ref (length a) - 1 in
while l ≤ u do
  let m = (l + u) / 2 in
  ...
```

$l + u$ may overflow with large arrays!

**Goal**
prove that a program is safe with respect to overflows
Target Type: int32

- 32-bit signed integers in two-complement representation: integers between $-2^{31}$ and $2^{31} - 1$.

- If the mathematical result of an operation fits in that range, that is the computed result.

- Otherwise, an overflow occurs. Behavior depends on language and environment: modulo arith, saturated arith, abrupt termination, etc.

A program is safe if no overflow occurs.
Idea: replace all arithmetic operations by abstract functions with preconditions. $x + y$ becomes \texttt{int32\_add}(x, y).

\begin{verbatim}
val int32_add(x: int, y: int): int
  requires -2^{31} \leq x + y < 2^{31}
  ensures result = x + y
\end{verbatim}

Unsatisfactory: range constraints of integer must be added explicitly everywhere
Idea: replace

- type `int` with an abstract type `int32` coercible to it,
- all operations by abstract functions with preconditions,
and add an axiom about the range of `int32`.

\[
\text{predicate } \text{in_int32} (n: \text{int}) = -2^{31} \leq n < 2^{31}
\]

\textbf{type} \texttt{int32}

\textbf{function} \texttt{of_int32}(x: \texttt{int32}): \texttt{int}

\textbf{axiom} \texttt{bounded_int32}: \texttt{forall} x: \texttt{int32}. \texttt{in_int32}(\texttt{of_int32}(x))

\textbf{val} \texttt{int32_add}(x: \texttt{int32}, y: \texttt{int32}): \texttt{int32}

\textbf{requires} \texttt{in_int32}(\texttt{of_int32}(x) + \texttt{of_int32}(y))

\textbf{ensures} \texttt{of_int32}(\texttt{result}) = \texttt{of_int32}(x) + \texttt{of_int32}(y)
Binary Search with overflow checking

See `bin_search_int32.mlw`
Binary Search with overflow checking

See `bin_search_int32.mlw`

**Application**

Used for translating mainstream programming language into Why3:
- From C to Why3: Frama-c, Jessie plug-in
  See `bin_search.c`
- From Java to Why3: Krakatoa
- From Ada to Why3: Spark2014
Floating-Point Arithmetic

- Limited range $\Rightarrow$ exceptional behaviors.
- Limited precision $\Rightarrow$ inaccurate results.
Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.
Width: $1 + w_e + w_m = 32$, or $64$, or $128$.
Bias: $2^{w_e-1} - 1$. Precision: $p = w_m + 1$.

A floating-point datum

<table>
<thead>
<tr>
<th>sign $s$</th>
<th>biased exponent $e'$ ($w_e$ bits)</th>
<th>mantissa $m$ ($w_m$ bits)</th>
</tr>
</thead>
</table>

represents
## Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.

**Width:** $1 + w_e + w_m = 32$, or 64, or 128.

**Bias:** $2^{w_e-1} - 1$. **Precision:** $p = w_m + 1$.

A floating-point datum

<table>
<thead>
<tr>
<th>sign $s$</th>
<th>biased exponent $e'$ ($w_e$ bits)</th>
<th>mantissa $m$ ($w_m$ bits)</th>
</tr>
</thead>
</table>

represents

- if $0 < e' < 2^{w_e} - 1$, the real $(-1)^s \cdot 1.m' \cdot 2^{e' - bias}$, normal
- if $e' = 0$,
  - $\pm 0$ if $m' = 0$, zeros
  - the real $(-1)^s \cdot 0.m' \cdot 2^{-bias+1}$ otherwise, subnormal
- if $e' = 2^{w_e} - 1$,
  - $(-1)^s \cdot \infty$ if $m' = 0$, infinity
  - *Not-a-Number* otherwise. NaN
## Floating-Point Data

\[ (-1)^s \times 2^{e-B} \times 1.f \]

\[ (-1)^1 \times 2^{198-127} \times 1.10010011110000111000000_2 \]

\[-2^{54} \times 206727 \approx -3.7 \times 10^{21}\]
IEEE-754 standard

A floating-point operator shall behave as if it was first computing the infinitely-precise value and then rounding it so that it fits in the destination floating-point format.

Rounding of a real number $x$:

Overflows are not considered when defining rounding: exponents are supposed to have no upper bound!
Specifications, main ideas

Same as with integers, we specify FP operations so that no overflow occurs.

constant max : real = 0x1.FFFFFEp127
predicate in_float32 (x:real) = abs x ≤ max
type float32
function of_float32(x: float32): real
axiom float32_range: forall x: float32. in_float32 (of_float32 x)

function round32(x: real): real
(* ... axioms about round32 ... *)

val float32_add(x: float32, y: float32): float32
  requires in_float32(round32(of_float32 x - of_float32 y))
  ensures of_float32 result = round32 (of_float32 x - of_float32 y)
Specifications in practice

- Several possible rounding modes
- Many axioms for `round32`, but incomplete anyway

Demo: `clock_drift.c`
Outline

Exercises from last lecture

Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations

Aliasing Issues
  Call by Reference
  Pointer Programs
Reminder of former lectures

Compound data structures can be modeled using expressive specification languages

- Defined functions and predicates
- Product types (records)
- Sum types (lists, trees)
- Axiomatizations (arrays, sets)

Important points:

- **pure** types, no internal “in-place” assignment
- Mutable variables = references to pure types
Aliasing

*Aliasing* = two different “names” for the same mutable data

Two sub-topics:

- Call by reference
- Pointer programs
Need for call by reference

Example: stacks of integers

```ml
type stack = list int

val s : ref stack

let fun push(x:int):unit
  writes s
  ensures s = Cons(x,s@Old)
  body ...

let fun pop(): int
  requires s \neq Nil
  writes s
  ensures result = head(s@Old) \land s = tail(s@Old)
```
Need for call by reference

If we need two stacks in the same program:

- We don’t want to write the functions twice!

We want to write

```plaintext
type stack = list int

let fun push(s:ref stack, x:int): unit
  writes s
  ensures s = Cons(x, s@Old)
  ...

let fun pop(s:ref stack): int
  ...
```
Call by Reference: example

```ml
val s1, s2 : ref stack

let fun test():
  writes s1, s2
  ensures result = 13 ∧ head(s2) = 42
  body push(s1,13); push(s2,42); pop(s1)
```

See file stack1.mlw
Aliasing problems

let fun test(s3,s4: ref stack) : unit
  writes s3, s4
  ensures { head(s3) = 42 ∧ head(s4) = 13 }
  body push(s3,42); push(s4,13)

let fun wrong(s5: ref stack) : int
  writes s5
  ensures { head(s5) = 42 ∧ head(s5) = 13 }
      something’s wrong !?
  body test(s5,s5)

Aliasing is a major issue

Deductive Verification Methods like Hoare logic, Weakest Precondition Calculus implicitly require absence of aliasing
Syntax

- Declaration of functions: (references first for simplicity)

  ```ml
  let fun f(y_1 : ref \tau_1, \ldots, y_k : ref \tau_k, x_1 : \tau'_1, \ldots, x_n : \tau'_n):
    \ldots
  ```

- Call:

  ```ml
  f(z_1, \ldots, z_k, e_1, \ldots, e_n)
  ```

  where each \( z_i \) must be a reference
Operational Semantics

Intuitive semantics, by substitution:

\[ \Pi' = \{ x_i \leftarrow [t_i]_{\Sigma, \Pi}, \} \quad \Sigma, \Pi' \models Pre \quad Body' = Body[y_j \leftarrow z_j] \]

\[ \Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \leadsto \Sigma, \Pi, (\text{Old} : \text{frame}(\Pi', \text{Body}', \text{Post})) \]

- The body is executed, where each occurrence of reference parameters are replaced by the corresponding reference argument.
- Not a “practical” semantics, but that’s not important...
Operational Semantics

Variant: Semantics by copy/restore:

\[
\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow \llbracket t_i \rrbracket\Sigma,\Pi\} \quad \Sigma, \Pi' \models Pre
\]

\[
\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \leadsto \Sigma', \Pi, (Old : frame(\Pi', Body, Post))
\]

\[
\Sigma, \Pi' \models P[result \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]
\]

\[
\Sigma, \Pi, (frame(\Pi', v, P)) \leadsto \Sigma', \Pi, v
\]
Operational Semantics

Variant: Semantics by copy/restore:

\[ \Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{ x_i \leftarrow [t_i]_{\Sigma, \Pi} \} \quad \Sigma, \Pi' \models Pre \]

\[ \Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \leadsto \Sigma', \Pi, (Old : frame(\Pi', Body, Post)) \]

\[ \Sigma, \Pi' \models P[result \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)] \]

\[ \Sigma, \Pi, (frame(\Pi', v, P)) \leadsto \Sigma', \Pi, v \]

Warning: not the same semantics!
Difference in the semantics

```ocaml
val g : ref int

let fun f(x:ref int):unit
  body x := 1; x := g+1

let fun test():unit
  body g:=0; f(g)
```

After executing test:

- Semantics by substitution: \( g = 2 \)
- Semantics by copy/restore: \( g = 1 \)
Aliasing Issues (1)

```ocaml
let fun f(x:ref int, y:ref int): writes x, y
  ensures x = 1 ∧ y = 2
  body x := 1; y := 2

val g : ref int

let fun test():
  body
    f(g,g);
    assert g = 1 ∧ g = 2 (⋆ ???? ⋆)
```

- Aliasing of reference parameters
Aliasing Issues (2)

val g1 : ref int
val g2 : ref int

let fun p(x:ref int):
  writes g1, x
  ensures g1 = 1 ∧ x = 2
  body g1 := 1; x := 2

let fun test():
  body
  p(g2); assert g1 = 1 ∧ g2 = 2; (* OK *)
p(g1); assert g1 = 1 ∧ g1 = 2; (* ??? *)

- Aliasing of a global variable and reference parameter
Aliasing Issues (3)

```ocaml
val g : ref int

val fun f(x:ref int):unit
  writes x
  ensures x = g + 1
  (* body x := 1; x := g+1 *)

let fun test():unit
  ensures { g = 1 or 2 ? }
  body g := 0; f(g)
```

- Aliasing of a read reference and a written reference
Aliasing Issues (3)

New need in specifications
Need to specify read references in contracts

```ocaml
val g : ref int

val f(x:ref int):unit
  reads g       (* new clause in contract *)
  writes x
  ensures x = g + 1
  (* body x := 1; x := g+1 *)

let fun test():unit
  ensures { g = ? }
  body g := 0; f(g)
```

▶ See file stack2.mlw
Typing: Alias-Freedom Conditions

For a function of the form

\[ f(y_1 : \tau_1, ..., y_k : \tau_k, ...) : \tau : \]

writes \( \vec{w} \)

reads \( \vec{r} \)

Typing rule for a call to \( f \):

\[
\begin{align*}
\ldots & \quad \forall ij, i \neq j \rightarrow z_i \neq z_j \\
& \quad \forall i, j, z_i \neq w_j \\
& \quad \forall i, j, z_i \neq r_j \\
\ldots \vdash f(z_1, \ldots, z_k, ...) : \tau
\end{align*}
\]

- effective arguments \( z_j \) must be distinct
- effective arguments \( z_j \) must not be read nor written by \( f \)
Thanks to restricted typing:
- Semantics by substitution and by copy/restore coincide
- Hoare rules remain correct
- WP rules remain correct
New references

- Need to return newly created references
- Example: stack continued

```plaintext
let fun create(): ref stack
    ensures result = Nil
    body (ref Nil)
```

- Typing should require that a returned reference is always \textit{fresh}

See file \texttt{stack3.mlw}
Pointer programs

- We drop the hypothesis “no reference to reference”
- Allows to program on *linked data structures*. Example (in the C language):

```c
struct List { int data; list next; } *list;
while (p <> NULL) { p->data++; p = p->next }
```

- “In-place” assignment
- References are now *values* of the language: “pointers” or “memory addresses”

We need to handle aliasing problems differently
Syntax

- For simplicity, we assume a language with pointers to records
  - Access to record field: $e \rightarrow f$
  - Update of a record field: $e \rightarrow f := e'$
Operational Semantics

- New kind of values: `loc` = the type of pointers
- A special value `null` of type `loc` is given
- A program state is now a pair of
  - a `store` which maps variables identifiers to values
  - a `heap` which maps pairs `(loc, field name)` to values
- Memory access and updates should be proved safe (no “null pointer dereferencing”)
- For the moment we forbid allocation/deallocation
Component-as-array trick

If:

▶ a program is well-typed
▶ The set of all field names are known

then the heap can be also seen as a finite collection of maps, one for each field name

▶ map for a field of type $\tau$ maps loc to values of type $\tau$

This “trick” allows to encode pointer programs into Why3 programs

▶ Use maps indexed by locs instead of integers
Component-as-array model

type loc
constant null : loc

val acc(field: ref (map loc α),l:loc) : α
  requires l ≠ null
  reads field
  ensures result = select(field,l)

val upd(field: ref (map loc α),l:loc,v:α):unit
  requires l ≠ null
  writes field
  ensures field = store(field@Old,l,v)

Encoding:
- Access to record field: \( e \rightarrow f \) becomes acc(f,e)
- Update of a record field: \( e \rightarrow f := e' \) becomes upd(f,e,e')
Example

- In C

```c
struct List { int data; list next; } *list;

while (p <> NULL) { p->data++; p = p->next }
```

- In Why3

```why3
val data: ref (map loc int)
val next: ref (map loc loc)

while p ≠ null do
    upd(data,p,acc(data,p)+1);
    p := acc(next,p)
```
In-place List Reversal

A la C/Java:

```c
list reverse(list l) {
    list p = l;
    list r = null;
    while (p != null) {
        list n = p->next;
        p->next = r;
        r = p;
        p = n
    }
    return r;
}
```
In-place Reversal in our Model

let fun reverse (l:loc) : loc =
  let p = ref l in
  let r = ref null in
  while (p ≠ null) do
    let n = acc(next,p) in
    store(next,p,r);
    r := p;
    p := n
  done;
  r

Goals:
  ▶ Specify the expected behavior of reverse
  ▶ Prove the implementation
Specifying the function

Predicate \( \text{list}_\text{seg}(p, \text{next}, p_M, q) : p \) points to a list of nodes \( p_M \) that ends at \( q \).

\[
p = p_0 \xrightarrow{\text{next}} p_1 \xrightarrow{\text{next}} \cdots \xrightarrow{\text{next}} p_k \xrightarrow{\text{next}} q
\]

\[
p_M = \text{Cons}(p_0, \text{Cons}(p_1, \cdots \text{Cons}(p_k, \text{Nil}) \cdots ))
\]

\( p_M \) is the \textit{model list} of \( p \)

\[
\text{inductive list}_\text{seg}(\text{loc}, \text{map loc loc}, \text{list loc loc}, \text{loc}) = \\
| \text{list}_\text{seg}_\text{nil}: \\
\quad \text{forall } p:\text{loc}, \text{next}:\text{map loc loc}. \text{list}_\text{seg}(p,\text{next},\text{Nil},p) \\
| \text{list}_\text{seg}_\text{cons}: \\
\quad \text{forall } p \ q:\text{loc}, \text{next}:\text{map loc loc}, p_M:\text{list loc}. \\
\quad p \neq \text{null} \land \text{list}_\text{seg}(\text{select}(\text{next},p),\text{next},p_M,q) \rightarrow \\
\quad \text{list}_\text{seg}(p,\text{next},\text{Cons}(p,pM),q)
\]
Specification

- pre: input \( l \) well-formed:
  \[ \exists l_M. \text{list\_seg}(l, \text{next}, l_M, \text{null}) \]

- post: output well-formed:
  \[ \exists r_M. \text{list\_seg}(\text{result}, \text{next}, r_M, \text{null}) \]

and

\[ r_M = \text{rev}(l_M) \]

Issue: quantification on \( l_M \) is global to the function

- Use *ghost* variables
Annotated In-place Reversal

```plaintext
let fun reverse (l:loc) (ghost lM:list loc) : loc =
  requires list_seg(l,next,lM,null)
  writes next
  ensures list_seg(result,next,rev(lM),null)
body
  let p = ref l in
  let r = ref null in
  while (p ≠ null) do
    let n = acc(next,p) in
    store(next,p,r);
    r := p;
    p := n
  done;
  r
```
In-place Reversal: loop invariant

```plaintext
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
p := n
```

Local ghost variables

\[
\text{list}_\text{seg}(p, next, p, null) = \text{list}_\text{seg}(r, next, r, null)
\]

\[\text{append}(\text{rev}(p), r) = \text{rev}(l)\]

See file linked_list_rev.mlw
In-place Reversal: loop invariant

while (p \neq \text{null}) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n

Local ghost variables $p_M, r_M$

\[ \text{list\_seg}(p, next, p_M, \text{null}) \]

\[ \text{list\_seg}(r, next, r_M, \text{null}) \]
In-place Reversal: loop invariant

```plaintext
while (p ≠ null) do
    let n = acc(next,p) in
    store(next,p,r);
    r := p;
    p := n
```

Local ghost variables $p_M, r_M$

- $\text{list\_seg}(p, next, p_M, null)$
- $\text{list\_seg}(r, next, r_M, null)$

$\text{append}(\text{rev}(p_M), r_M) = \text{rev}(l_M)$

See file `linked_list_rev.mlw`
Needed lemmas

- Need to show that `list_seg` remains true when `next` is updated:

```plaintext
lemma list_seg_frame: forall next1 next2:map loc loc, p q v: loc, pM:list loc.
  list_seg(p,next1,pM,null) \land
  next2 = store(next1,q,v) \land
  \neg mem(q,pM) \rightarrow list_seg(p,next2,pM,null)
```

- For that, need to show that $p_M, r_M$ are *disjoint*
- For that, need to show that a rep list do not contain repeated elements

```plaintext
lemma list_seg_no_repet:
  forall next:map loc loc, p: loc, pM:list loc.
  list_seg(p,next,pM,null) \rightarrow no_repet(pM)
```
Exercise

The algorithm that appends two lists \textit{in place} follows this pseudo-code:

\begin{verbatim}
append(l1,l2 : loc) : loc
  if l1 is empty then return l2;
  let ref p = l1 in
  while p→next is not null do p := p→next;
  p → next := l2;
  return l1
\end{verbatim}

1. Specify a post-condition giving the list models of both \texttt{result} and \texttt{l2} (add any ghost variable needed)

2. Which pre-conditions and loop invariants are needed to prove this function?

See \texttt{linked_list_app.mlw}
Reasoning on pointer programs using the component-as-array trick is complex
  ▶ need to state and prove \textit{frame} lemmas
  ▶ need to specify many \textit{disjointness} properties
  ▶ even harder is the handling of \textit{memory allocation}

\textit{Separation Logic} is another approach to reason on heap memory
  ▶ memory resources \textit{explicit} in formulas
  ▶ frame lemmas and disjointness properties are internalized