A Verified Implementation of the Bounded List Container

Raphaël Cauderlier, Mihaela Sighireanu

IRIF, University Paris Diderot, CNRS

ETAPS 2018, Thessaloniki, April 16
Verified Containers

- Good data structures are crucial for efficient programs
- Containers can be specified using mathematical models
- Not much work yet on functional verification of real world containers
Verified Containers

- Good data structures are crucial for efficient programs
- Containers can be specified using mathematical models
- Not much work yet on functional verification of real world containers
  - Pointer-level optimisations
  - Concurrency
  - Many theories to combine: arithmetics, sets, multisets, arrays, lists, etc...
This Work

Case study on a container library from the Ada standard library.

- **Given:**
  - Optimized Ada implementation (~ 1400 loc)
  - Tested SPARK specification (~ 3600 loc)
  - Used to verify safety-critical software

- **Done:**
  - Translation in C (~ 500 loc)
  - Specification in VeriFast (~ 1220 loc)
  - Verification in VeriFast (~ 3500 loc)
Outline

1. Bounded Doubly-Linked Lists
2. Verification
3. Conclusion
List is the type of Bounded Doubly-Linked Lists.

Capacity(List) : NonNegative
Empty_List(NonNegative) : List
Length(List) : NonNegative
=(List, List) : Boolean
Is_Empty(List) : Boolean

Clear(List)
Assign(List, List)
Copy(List, NonNegative) : List
Cursor is the type of positions inside lists.

No_Element : Cursor
First(List) : Cursor Last(List) : Cursor
Next(List, Cursor) Previous(List, Cursor)
Element(List, Cursor) : Element_Type
Find(List, Element_Type, Cursor) : Cursor

Replace_Element(List, Cursor, Element_Type)
Insert(List, Cursor, Element_Type)
Delete(List, Cursor)
...
Implementation: Nodes

A Node is a record with the following fields:

- **Element**: Element_Type
- **Prev**: -1..Capacity
- **Next**: 0..Capacity

A node is *free* if Prev = -1, otherwise it is *occupied*.

![Node Diagram]

---

**Diagram:**

- The first node has Prev = 1, Element = 1, and Next = 0, indicating it is not free.
- The second node has Prev = -1, Element = 1, and Next = 0, indicating it is free.
Implementation: Lists

A List is a record with the following fields:

- **Nodes[1..Capacity]**: an array of nodes
- **Length**: 0..Capacity
- **Free**: -Capacity..Capacity
- **First**: 0..Capacity
- **Last**: 0..Capacity

When Free ≥ 0, we call the list initialized.
A List is a record with the following fields:

- **Nodes[1..Capacity]**: an array of nodes
- **Length**: 0..Capacity
- **Free**: -Capacity..Capacity
- **First**: 0..Capacity
- **Last**: 0..Capacity

When Free $\geq 0$, we call the list *initialized*.

Valid cursors are indexes of occupied nodes.

No_Element := 0
Implicit Invariants:

- Occupied nodes form a doubly-linked list of length Length between Nodes[First] and Nodes[Last].

- If the list is initialized, then free nodes form a simply-linked list from Free to 0.

- Otherwise, free nodes are the nodes Nodes[-Free], Nodes[-Free+1], ..., Nodes[Capacity].
Implicit Invariants:

- Occupied nodes form a doubly-linked list of length \( \text{Length} \) between \( \text{Nodes}[\text{First}] \) and \( \text{Nodes}[\text{Last}] \).

- If the list is initialized, then free nodes form a simply-linked list from \( \text{Free} \) to 0.

- Otherwise, free nodes are the nodes \( \text{Nodes}[\text{-Free}] \), \( \text{Nodes}[\text{-Free+1}] \), ..., \( \text{Nodes}[\text{Capacity}] \).
Implementation: Lists

Implicit Invariants:

- Occupied nodes form a doubly-linked list of length \( \text{Length} \) between \( \text{Nodes[First]} \) and \( \text{Nodes[Last]} \).

- If the list is initialized, then free nodes form a simply-linked list from \( \text{Free} \) to 0.

- Otherwise, free nodes are the nodes \( \text{Nodes[-Free]} \), \( \text{Nodes[-Free+1]} \), ..., \( \text{Nodes[Capacity]} \).

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prev</td>
<td>Elem</td>
<td>Next</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
```
Example

Capacity: 5
Length: 0
Free: -1
First: 0
Last: 0

$L = \text{Empty\_List}(5)$
Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>●</td>
<td>0</td>
<td>-1</td>
<td>●</td>
</tr>
</tbody>
</table>

Capacity: 5
Length: 0
Free: -1
First: 0
Last: 0

Insert(L, Last(L), e1)
Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Capacity: 5  
Length: 1  
Free: -2  
First: 1  
Last: 1

Insert(L, Last(L), e1)
Example

Capacity: 5
Length: 1
Free: -2
First: 1
Last: 1

Insert(L, Last(L), e2)
Insert(L, Last(L), e3)
Example

Capacity: 5
Length: 3
Free: -4
First: 1
Last: 3

Insert(L, Last(L), e2)
Insert(L, Last(L), e3)
Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prev</td>
<td>Elem</td>
<td>Next</td>
<td>Prev</td>
<td>Elem</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Capacity: 5
Length: 3
Free: -4
First: 1
Last: 3

\[c = \text{First}(L)\]
\[\text{Next}(c)\]
\[\text{Delete}(L, c)\]
Example

Nodes[1]

<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Nodes[2]

<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Nodes[3]

<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Nodes[4]

<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>

Nodes[5]

<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Capacity: 5
Length: 2
Free: 2
First: 1
Last: 3

c = First(L)
Next(c)
Delete(L, c)
Example

Capacity: 5
Length: 2
Free: 2
First: 1
Last: 3
Example

Capacity: 5
Length: 2
Free: 2
First: 1
Last: 3

Nodes[1]
<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Nodes[2]
<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>4</td>
<td>-1</td>
</tr>
</tbody>
</table>

Nodes[3]
<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Nodes[4]
<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>5</td>
<td>-1</td>
</tr>
</tbody>
</table>

Nodes[5]
<table>
<thead>
<tr>
<th>Prev</th>
<th>Elem</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Insert(L, Last(L), e4)
Example

Capacity: 5
Length: 3
Free: 4
First: 1
Last: 2

Insert(L, Last(L), e4)
Each method of the library is specified by its impact on Model and Positions.

Model(List) : Sequence
Positions(List) : Map(Cursor, Positive)
procedure Append

( Container : in out List;
  New_Item  : Element_Type;
  Count     : Count_Type)

with

Global => null,
Pre   =>
  Length (Container) <= Container.Capacity - Count,
Specification

Post =>
Length (Container) = Length (Container)'Old + Count
and Model (Container)'Old <= Model (Container)
and (if Count > 0 then
  M.Constant_Range
  (Container => Model (Container),
   Fst => Length (Container)'Old + 1,
   Lst => Length (Container),
   Item => New_Item))
and P_Positions_Truncated
  (Positions (Container)'Old,
   Positions (Container),
   Cut => Length (Container)'Old + 1,
   Count => Count);
VeriFast:

- Verification tool for C (and Java)
- Specification language: separation logic with data types and inductive predicates
- Heuristics
- Backend: SMT Solvers (Redux, Z3)
The Ada library has been manually translated in C and VeriFast.

- 0-starting arrays
- Capacity becomes a field of `List`
- Contract cases become alternatives
- Distinct languages for programming and specification
  - Functional models cannot exist at runtime
  - Functional and imperative lists are no more two instances of the same interface
VeriFast Logic

- **Quantifier-free Separation Logic:**

  \[
  t ::= x \mid f(t_1, \ldots, t_n) \\
  \varphi ::= \text{emp} \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \ast \varphi_2 \mid P(t_1, \ldots, t_n)
  \]
VeriFast Logic

- Quantifier-free Separation Logic:
  
  \[
  t \ ::= \ x \mid f(t_1, \ldots, t_n) \\
  \varphi \ ::= \ emp \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n)
  \]

- Algebraic Data Types

  Example: sequence \( \langle a \rangle \ ::= \text{Nil} \mid \text{Cons of } a \star \text{sequence } \langle a \rangle \)

  Functions defined by structural recursion
VeriFast Logic

- Quantifier-free Separation Logic:
  \[
  t \ ::= \ x \mid f(t_1, \ldots, t_n) \\
  \varphi \ ::= \ \text{emp} \mid t_1 = t_2 \mid t_1 \leftrightarrow t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n)
  \]

- Algebraic Data Types
  Example: sequence \( \langle a \rangle \) := Nil \mid Cons \ of \ a \ast sequence \( \langle a \rangle \)
  Functions defined by structural recursion

- Inductive predicates
  Example:
  \[\text{linked\_list}(x, y) := (x = y) \mid \exists z. x \leftrightarrow z \ast \text{linked\_list}(z, y)\]
VeriFast Logic

- **Quantifier-free Separation Logic:**
  \[
  t ::= x \mid f(t_1, \ldots, t_n) \mid \{ l_1 = t_1, \ldots, l_n = t_n \} \mid t.l \mid t_1 + t_2 \\
  \varphi ::= \text{emp} \mid t_1 = t_2 \mid t_1 \not\rightarrow t_2 \mid \varphi_1 \ast \varphi_2 \mid P(t_1, \ldots, t_n)
  \]

- **Algebraic Data Types**
  Example: sequence \( \langle a \rangle \) := Nil | Cons of a \ast sequence \( \langle a \rangle \)
  Functions defined by structural recursion

- **Inductive predicates**
  Example:
  \[
  \text{linked}_\text{list}(x, y) := (x = y) \mid \exists z . x \not\rightarrow z \ast \text{linked}_\text{list}(z, y)
  \]
Low-level invariants

\[
\begin{align*}
\text{free\_range}(\text{Nodes}, \text{first}, \text{last}) &:= \text{first} = \text{last} \\
&\mid \exists X. \text{Nodes} + \text{first} \mapsto \{\text{Prev} = -1, \text{Elem} = X, \text{Next} = 0\} \\
&\quad \star \text{free\_range}(\text{Nodes}, \text{first} + 1, \text{last})
\end{align*}
\]

\[
\begin{align*}
\text{free\_sll}(\text{Nodes}, \text{first}, \text{last}) &:= \text{first} = \text{last} \\
&\mid \exists n, X. \text{Nodes} + \text{first} \mapsto \{\text{Prev} = -1, \text{Elem} = X, \text{Next} = n\} \\
&\quad \star \text{free\_sll}(\text{Nodes}, n, \text{last})
\end{align*}
\]

\[
\begin{align*}
\text{dll}(\text{Nodes}, \text{prev}, \text{from}, \text{last}, \text{to}) &:= \text{prev} = \text{last} \star \text{from} = \text{to} \\
&\mid \exists n, X. \text{Nodes} + \text{from} \mapsto \{\text{Prev} = \text{prev}, \text{Elem} = X, \text{Next} = n\} \\
&\quad \star \text{dll}(\text{Nodes}, \text{from}, n, \text{last}, \text{to})
\end{align*}
\]

\[
\begin{align*}
\text{bdll}(L) &:= \text{dll}(L.n\text{odes}, 0, L.first, L.last, 0) \star \\
&\quad (L.free < 0 \star \text{free\_range}(L.n\text{odes}, -\text{free}, L.capacity) \\
&\mid L.free > 0 \star \text{free\_sll}(L.n\text{odes}, \text{free}, 0))
\end{align*}
\]
High-level models

\[
\text{sequence} \langle a \rangle := \text{Nil} | \text{Cons of} \ a \ast \text{sequence} \langle a \rangle \\
\text{prod} \langle a, b \rangle := \text{Pair of} \ a \ast b \\
\text{map} \langle a, b \rangle := \text{sequence} \langle \text{prod} \langle a, b \rangle \rangle
\]
High-level models

\[\text{sequence } \langle a \rangle := \text{Nil } \mid \text{Cons of } a \ast \text{sequence } \langle a \rangle\]

\[\text{prod } \langle a, b \rangle := \text{Pair of } a \ast b\]
\[\text{map } \langle a, b \rangle := \text{sequence } \langle \text{prod } \langle a, b \rangle \rangle\]

+ many unusual relations
Precise models

\[
dll(Nodes, prev, from, last, to) :=
\begin{align*}
&\text{prev} = \text{last} \star \text{from} = \text{to} \\
&\exists n, X . \\
&\text{Nodes} + \text{from} \mapsto \{ \text{Prev} = \text{prev}, \text{Elem} = X, \text{Next} = n \}
\star dll(Nodes, from, n, last, to)
\end{align*}
\]
Precise models

\[
\text{precise\_model} \quad ::= \quad C_0 \mid C_1
\]

\[
dll(Nodes, \text{prev}, \text{from}, \text{last}, \text{to}, m) \quad ::= \\
\mid \quad \text{prev} = \text{last} \star \text{from} = \text{to} \star m = C_0 \\
\mid \exists n, X, m'. \\
\quad Nodes + \text{from} \mapsto \{ \text{Prev} = \text{prev}, \text{Elem} = X, \text{Next} = n \} \\
\star dll(Nodes, \text{from}, n, \text{last}, \text{to}, m') \\
\star m = C_1
\]
Precise models

\[
\text{precise\_model} \langle a \rangle := C_0 | C_1 \text{ of int} \star a \star \text{precise\_model} \langle a \rangle
\]

\[
dll(\text{Nodes}, \text{prev}, \text{from}, \text{last}, \text{to}, m) :=
\]
\[
| \quad \text{prev} = \text{last} \star \text{from} = \text{to} \star m = C_0
\]
\[
| \exists n, X, m'.
\]
\[
\text{Nodes} + \text{from} \mapsto \{ \text{Prev} = \text{prev}, \text{Elem} = X, \text{Next} = n \}
\star dll(\text{Nodes}, \text{from}, n, \text{last}, \text{to}, m')
\star m = C_1(n, X, m')
\]
Precise models

\[
\text{precise\_model} \ \langle a \rangle := C_0 \mid C_1 \text{ of int} \ast a \ast \text{precise\_model} \langle a \rangle
\]

\[
\text{dll}(\text{Nodes}, \text{prev}, \text{from}, \text{last}, \text{to}, m) := \text{match } m \text{ with}
\]
\[
| C_0 \rightarrow \text{prev} = \text{last} \ast \text{from} = \text{to}
\]
\[
| C_1(n, X, m') \rightarrow
\]
\[
\text{Nodes} + \text{from} \mapsto \{ \text{Prev} = \text{prev}, \text{Elem} = X, \text{Next} = n \}
\]
\[
\ast \text{dll}(\text{Nodes}, \text{from}, n, \text{last}, \text{to}, m')
\]
precise_model  ⟨a⟩ := C₀ | C₁ of int * a * precise_model  ⟨a⟩

dll(Nodes, prev, from, last, to, m) := match m with
  | C₀ → prev = last * from = to
  | C₁(n, X, m') →
    Nodes + from ↑ { Prev = prev, Elem = X, Next = n }
    * dll(Nodes, from, n, last, to, m')

- From a precise model, high-level models are easy to define
Precise models

\[
\text{precise\_model} \; \langle a \rangle := C_0 \mid C_1 \; \text{of} \; \text{int} \ast a \ast \text{precise\_model} \; \langle a \rangle
\]

\[
dl\langle Nodes, prev, from, last, to, m \rangle := \text{match} \; m \; \text{with}
\]
\[
| \; C_0 \rightarrow prev = last \ast from = to
\]
\[
| \; C_1 (n, X, m') \rightarrow
\]
\[
\text{Nodes} + from \mapsto \{ \text{Prev} = prev, \text{Elem} = X, \text{Next} = n \}
\]
\[
\ast \; dll\langle Nodes, from, n, last, to, m' \rangle
\]

- From a precise model, high-level models are easy to define
- Precise model compose well
Results

- 27 proved methods
  Remaining: sorting functions and Copy
- 47 inductive predicates, 42 pure recursive functions, 171 lemmata
- In Ada/SPARK: 1 implementation loc. for 3 specification loc.
- In VeriFast: 1 implementation loc. for 3 spec and 7 verif loc.
- Verification time: 1.7s
- Verification effort: 5 man-months
- 0 bugs found
Conclusion

- Challenges dealt:
  - contracts in use and complete
  - contracts refined with precise model
  - efficient code
  - reusable formal libs for sequences & maps

- VeriFast is a powerful tool
  - good automation for linear arithmetics
  - but no support for other theories
Future work

- More prover integration in VeriFast
- Automation of induction reasoning
- Remaining functions
- Verification of applications
References

- Ada/SPARK library: https://sourceware.org/svn/gcc/tags/gcc_7_1_0_release/gcc/ada/
- VeriFast: https://github.com/verifast/verifast
- Case study: https://doi.org/10.6084/m9.figshare.5919145