A Verified Implementation of the Bounded List Container

*Raphaël Cauderlier, Mihaela Sighireanu*

March 26, 2018
Outline

1. Introduction
2. Bounded Doubly-Linked Lists
3. Verification
4. Conclusion
Outline

1. Introduction
2. Bounded Doubly-Linked Lists
3. Verification
4. Conclusion
Verified Containers

- Good data structures are crucial for efficient programs
- Containers are usually easy to specify using mathematical models
- Not much work yet on verification of real world containers
Challenges

- Low-level reasoning on pointers
- Concurrency
- Optimisations
- 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)
  - SPARK specification (~ 3600 loc)

- **Done:**
  - Reimplementation in C (~ 600 loc)
  - Verification in VeriFast (~ 4700 loc)
Ada and SPARK

- **Ada**
  - General purpose, high-level programming language
  - Strong static typing
  - Generic

- **SPARK**
  - Subset of Ada with simple semantics
  - Executable contracts

Application to safety-critical embedded system
Ada Containers

- Lists, Vectors, Maps, Sets, and Graphs
- Purely functionnal or imperative
- Bounded or unbounded
- Generic in the element type
- Avoid most unnecessary pointer indirections
- Specified in SPARK, tested but not verified
- Not concurrent
Outline

1. Introduction
2. Bounded Doubly-Linked Lists
3. Verification
4. Conclusion
Interface

List is a type with the following interface:

Capacity : List -> NonNegative
Empty_List : List
Length : List -> NonNegative
= : List -> List -> Boolean
Is_Empty : List -> Boolean

Clear : List -> Unit
Assign : List -> List -> Unit
Copy : List -> NonNegative -> List

Model : List -> Sequence
Interface: Cursors

Cursor is a type with the following interface:

No_Element : Cursor
First, Last : List -> Cursor
Next, Previous : List -> Cursor -> Unit
Element : List -> Cursor -> Element_Type
Find : List -> Element_Type -> Cursor -> Cursor

Replace_Element : List -> Cursor -> Element_Type -> Unit
Insert : List -> Cursor -> Element_Type -> NonNegative -> Cursor
Delete : List -> Cursor -> NonNegative -> Unit

Positions : List -> Map(Cursor, Positive)
Specification

Each method of the library is specified by its impact on Model and Positions.
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);
A Node is a record with the following fields:

- **Element**: `Element_Type`
- **Prev**: `-1 ...` (Invariant: `Prev \leq Capacity`)
- **Next**: `NonNegative` (Invariant: `Next \leq Capacity`)

A node is **free** if `Prev = -1`, otherwise it is **occupied**.
Implementation: Lists

A List is a record with the following fields:

- **Nodes**: an array of Nodes of length Capacity
- **Length**: NonNegative (Invariant: Length ≤ Capacity)
- **Free**: Integer (Invariant: - Capacity ≤ Free ≤ Capacity)
- **First**: NonNegative (Invariant: First ≤ Capacity)
- **Last**: NonNegative (Invariant: Last ≤ Capacity)

When Free ≥ 0, we call the list initialized.
Implementation: Lists

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].
A cursor is either 0 (representing No_Element) or the index of an occupied node in the array Nodes.
Example

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

L = List(5)
### Example

<table>
<thead>
<tr>
<th>Capacity:</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length:</td>
<td>0</td>
</tr>
<tr>
<td>Free:</td>
<td>-1</td>
</tr>
<tr>
<td>First:</td>
<td>0</td>
</tr>
<tr>
<td>Last:</td>
<td>0</td>
</tr>
</tbody>
</table>

Append(L, e1, 1)

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

<table>
<thead>
<tr>
<th>Nodes[2]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prev</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Nodes[4]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Prev</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

Append(L, e1, 1)
Example

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

Append(L, e2, 1)
Example

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

Append(L, e2, 1)
Example

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

Append(L, e3, 1)
Example

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

Append(L, e3, 1)
Example

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

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

\[ \text{Delete}(L, c, 1) \]
Example

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

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

Delete(L, c, 1)
Example

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

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

Append(L, e4, 1)
Example

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

Append(L, e4, 1)
Example

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

Append(L, e5, 1)
Example

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

Append(L, e5, 1)
Outline

1. Introduction
2. Bounded Doubly-Linked Lists
3. Verification
4. Conclusion
VeriFast:

- Verification tool for C (and Java)
- Specification language: separation logic with data types and inductive predicates
- Backend: SMT Solvers (Redux, Z3)
Translation

The Ada library has been manually translated in C and VeriFast.

- 0-starting arrays
- Capacity becomes a field of List
- Strong language distinction between programming and specification
  - Functional models cannot exist at runtime
  - Functional and imperative lists are no more two instances of the same interface
- Contract cases
  - Translated to alternatives
VeriFast Logic

- Quantifier-free Separation Logic:

\[
\begin{align*}
    t & ::= x | f(t_1, \ldots, t_n) \\
    \varphi & ::= \text{emp} | t_1 = t_2 | t_1 \leftrightarrow t_2 | \varphi_1 \land \varphi_2 | P(t_1, \ldots, t_n)
\end{align*}
\]
VeriFast Logic

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

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

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

- Algebraic Data Types
  Example: sequence \( \langle a \rangle \) := Nil \mid Cons \textbf{of} \ a \star \text{sequence} \langle a \rangle
  Functions defined by structural recursion

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

- Quantifier-free Separation Logic:
  \[ t ::= x \mid f(t_1, \ldots, t_n) \mid \{l_1 = t_1, \ldots, l_n = t_n\} \mid t.l \mid t_1 + t_2 \]
  \[ \varphi ::= \text{emp} \mid t_1 = t_2 \mid t_1 \leftrightarrow t_2 \mid \varphi_1 \ast \varphi_2 \mid P(t_1, \ldots, t_n) \]

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

- Inductive predicates
  Example:
  \[ \text{linked_list}(x, y) ::= (x = y) \mid \exists z. x \leftrightarrow z \ast \text{linked_list}(z, y) \]
Low-level invariant

\[
\text{range}(\text{Nodes}, \text{first}, \text{last}) := \text{first} = \text{last} \\
| \exists X. \text{Nodes} + \text{first} \mapsto \{\text{Prev} = -1, \text{Elem} = X, \text{Next} = 0\} \\
* \text{range}(\text{Nodes}, \text{first} + 1, \text{last})
\]

\[
\text{sll}(\text{Nodes}, \text{first}, \text{last}) := \text{first} = \text{last} \\
| \exists n, X. \text{Nodes} + \text{first} \mapsto \{\text{Prev} = -1, \text{Elem} = X, \text{Next} = n\} \\
* \text{sll}(\text{Nodes}, n, \text{last})
\]

\[
\text{dll}(\text{Nodes}, \text{first}, \text{next}, \text{prev}, \text{last}) := \text{first} = \text{prev} * \text{next} = \text{last} \\
| \exists n, X. \text{Nodes} + \text{next} \mapsto \{\text{Prev} = \text{first}, \text{Elem} = X, \text{Next} = n\} \\
* \text{dll}(\text{Nodes}, \text{next}, n, \text{prev}, \text{last})
\]

\[
\text{bdll}(L) := \text{dll}(L.\text{nodes}, 0, L.\text{first}, L.\text{last}, 0) * \\
( L.\text{free} < 0 * \text{range}(L.\text{nodes}, -\text{free}, L.\text{capacity}) \\
| L.\text{free} > 0 * \text{sll}(L.\text{nodes}, \text{free}, 0) )
\]
High-level models

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

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

\[
\text{map} \langle a, b \rangle := \text{sequence} \langle \text{prod} \langle a, b \rangle \rangle
\]
Precise models

dll\((\text{Nodes}, \text{first}, \text{next}, \text{prev}, \text{last})\) :=
\begin{align*}
&\text{first} = \text{prev} \ast \text{next} = \text{last} \\
\text{\exists n, X}.
\end{align*}
Precise models

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

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

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

\[
dll(Nodes, first, next, prev, last, m) :=
\mid first = prev \times next = last \times m = C_0
\mid \exists n, X, m'.
\]

\[
Nodes + next \mapsto \{ \text{Prev} = first, \text{Elem} = X, \text{Next} = n\}
\times dll(Nodes, next, n, prev, last, m')
\times m = C_1(n, X, m')
\]
Precise models

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

\[
\text{dll}(\text{Nodes, first, next, prev, last, } m) := \text{match } m \text{ with}
\]
\[
| C_0 \rightarrow \text{first } = \text{prev } \ast \text{next } = \text{last}
\]
\[
| C_1(n, X, m') \rightarrow
\]
\[
\text{Nodes } + \text{next } \mapsto \{ \text{Prev } = \text{first, Elem } = X, \text{Next } = n \}
\]
\[
\ast \text{dll}(\text{Nodes, next, n, prev, last, } m')
\]
Precise models

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

\[
\text{model}(m) := \text{match } m \text{ with } \\
| C_0 \rightarrow \text{Nil} \\
| C_1(n, X, m') \rightarrow \text{Cons}(X, \text{model}(m'))
\]

\[
\text{positions}(m, \text{first}, i) := \text{match } m \text{ with } \\
| C_0 \rightarrow \text{Nil} \\
| C_1(n, X, m') \rightarrow \text{Cons}(\text{Pair}(\text{first}, i), \text{positions}(m', n, i + 1))
\]
Precise model composition

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

\[
\text{precise\_append} (m_1, m_2) := \textbf{match } m_1 \textbf{ with }
\mid C_0 \to m_2
\mid C_1(n, X, m') \to C_1(n, X, \text{precise\_append}(m', m_2))
\]

\[
dl(\text{Nodes, first, next, a, b, } m_1) \star \ndl(\text{Nodes, a, b, prev, last, } m_2) \vdash
\]

\[
dl(\text{Nodes, first, next, prev, last, } \text{precise\_append}(m_1, m_2))
\]

\[
dl(\text{Nodes, first, next, prev, last, } \text{precise\_append}(m_1, m_2)) \vdash
\]

\[
\exists a, b. \ndl(\text{Nodes, first, next, a, b, } m_1) \star \ndl(\text{Nodes, a, b, prev, last, } m_2)
\]
Results

- 27/39 proved methods Remaining: sorting functions and Copy
- 47 inductive predicates, 42 pure recursive functions, 171 lemmata
- In Ada/SPARK: 1 source code line for about 3 specification lines
- In Verifast: 1 source code line for about 8 annotation lines
Outline

1. Introduction
2. Bounded Doubly-Linked Lists
3. Verification
4. Conclusion
Conclusion

- Verifast is a powerful but limited tool
  - good automation for linear arithmetics
  - no support for other theories

- BDLL Library:
  - No error found
  - Static and dynamic assertions have been proved
  - Invariants made explicit
Future work

- Remaining functions
- More prover integration in VeriFast
- Automation of induction reasoning
- Transfer to other verification tools