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 ($\sim$ 1400 loc)
  - SPARK specification ($\sim$ 3600 loc)
- Done:
  - Reimplementation in C ($\sim$ 600 loc)
  - Verification in VeriFast ($\sim$ 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
Asssign : 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`.

Specification

```
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);
```

Implementation: Nodes

A Node is a record with the following fields:

- Element : Element_Type
- Prev : -1 ... (Invariant : Prev ≤ Capacity)
- Next : NonNegative (Invariant : Next ≤ 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` $\leq$ `Capacity`)
- `Free` : `Integer` (Invariant : - `Capacity` $\leq$ `Free` $\leq$ `Capacity`)
- `First` : `NonNegative` (Invariant : `First` $\leq$ `Capacity`)
- `Last` : `NonNegative` (Invariant : `Last` $\leq$ `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]`.

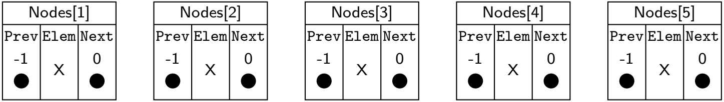
Cursors

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

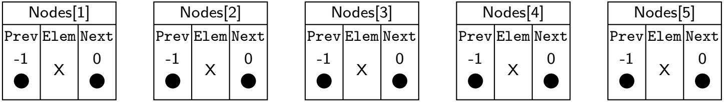
`L = List(5)`



Example

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

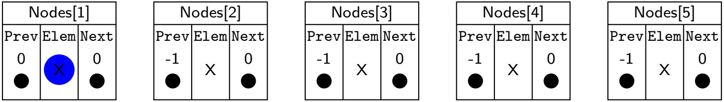
`Append(L, e1, 1)`



Example

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

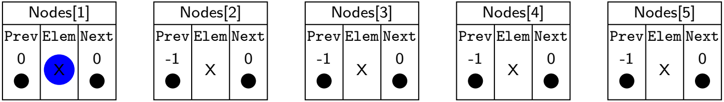
`Append(L, e1, 1)`



Example

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

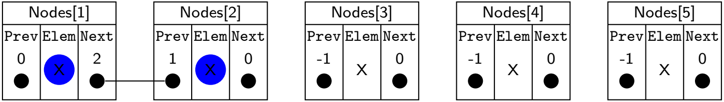
`Append(L, e2, 1)`



Example

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

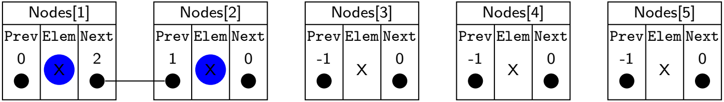
`Append(L, e2, 1)`



Example

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

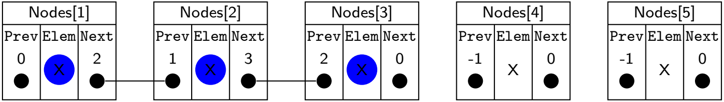
`Append(L, e3, 1)`



Example

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

`Append(L, e3, 1)`

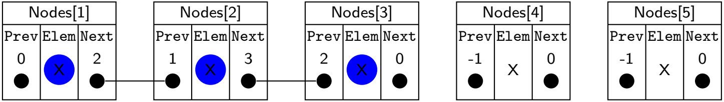


Example

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

```
c = Next(L,First(L))

Delete(L, c, 1)
```

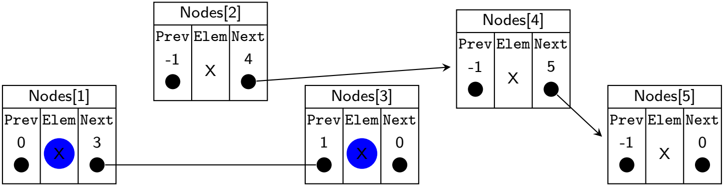


Example

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

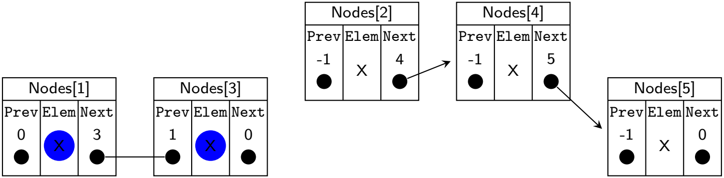
```
c = Next(L,First(L))

Delete(L, c, 1)
```



Example

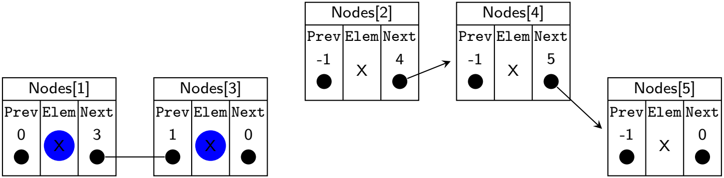
Capacity:	5
Length:	2
Free:	2
First:	1
Last:	3



Example

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

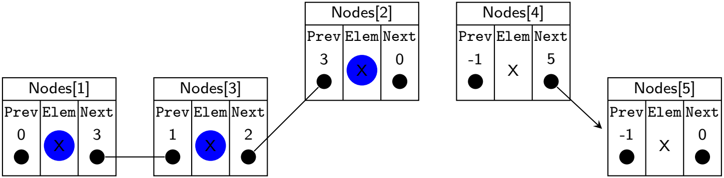
`Append(L, e4, 1)`



Example

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

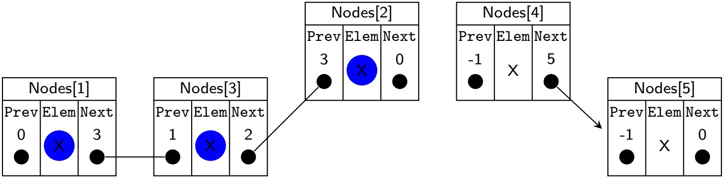
`Append(L, e4, 1)`



Example

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

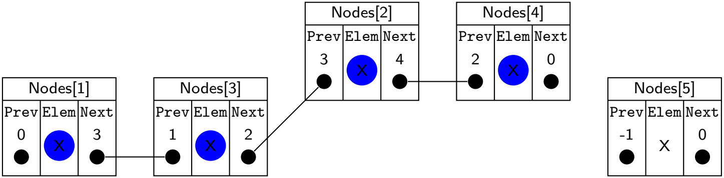
`Append(L, e5, 1)`



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

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 betwen 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:

$$
\begin{array}{lcl}
t & ::= & x \mid f(t_1, \ldots, t_n) \\
\varphi & ::= & \mathsf{emp} \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n)
\end{array}
$$

VeriFast Logic

- Quantifier-free Separation Logic:

$$\begin{array}{lcl} t & ::= & x \mid f(t_1, \ldots, t_n) \\ \varphi & ::= & \mathsf{emp} \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n) \end{array}$$

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

VeriFast Logic

- Quantifier-free Separation Logic:

$$\begin{array}{lcl} t & ::= & x \mid f(t_1, \ldots, t_n) \\ \varphi & ::= & \mathsf{emp} \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n) \end{array}$$

- Algebraic Data Types
  Example: sequence $\langle a \rangle$:= Nil | Cons **of** $a *$ sequence $\langle a \rangle$
  Functions defined by structural recursion
- Inductive predicates
  Example:
  linked_list(x, y) := (x = y) | $\exists z.\ x \mapsto z \star$ linked_list(z, y)

VeriFast Logic

- Quantifier-free Separation Logic:

$$
\begin{array}{lcl}
t & ::= & x \mid f(t_1, \ldots, t_n) \mid \{l_1 = t_1, \ldots, l_n = t_n\} \mid t.l \mid t_1 + t_2 \\
\varphi & ::= & \mathsf{emp} \mid t_1 = t_2 \mid t_1 \mapsto t_2 \mid \varphi_1 \star \varphi_2 \mid P(t_1, \ldots, t_n)
\end{array}
$$

- Algebraic Data Types
  Example: sequence $\langle a \rangle$:= Nil | Cons **of** $a *$ sequence $\langle a \rangle$
  Functions defined by structural recursion
- Inductive predicates
  Example:
  linked_list(x, y) := (x = y) | $\exists z.\ x \mapsto z \star$ linked_list(z, y)

Low-level invariant

$$\begin{aligned}&\mathrm{range}(\mathit{Nodes}, \mathit{first}, \mathit{last}) := \mathit{first} = \mathit{last}\\ &\mid \exists X.\ \mathit{Nodes} + \mathit{first} \mapsto \{\mathit{Prev} = -1, \mathit{Elem} = X, \mathit{Next} = 0\}\\ &\qquad \star\, \mathrm{range}(\mathit{Nodes}, \mathit{first} + 1, \mathit{last})\end{aligned}$$

$$\begin{aligned}&\mathrm{sll}(\mathit{Nodes}, \mathit{first}, \mathit{last}) := \mathit{first} = \mathit{last}\\ &\mid \exists n, X.\ \mathit{Nodes} + \mathit{first} \mapsto \{\mathit{Prev} = -1, \mathit{Elem} = X, \mathit{Next} = n\}\\ &\qquad \star\, \mathrm{sll}(\mathit{Nodes}, n, \mathit{last})\end{aligned}$$

$$\begin{aligned}&\mathrm{dll}(\mathit{Nodes}, \mathit{first}, \mathit{next}, \mathit{prev}, \mathit{last}) := \mathit{first} = \mathit{prev} \star \mathit{next} = \mathit{last}\\ &\mid \exists n, X.\ \mathit{Nodes} + \mathit{next} \mapsto \{\mathit{Prev} = \mathit{first}, \mathit{Elem} = X, \mathit{Next} = n\}\\ &\qquad \star\, \mathrm{dll}(\mathit{Nodes}, \mathit{next}, n, \mathit{prev}, \mathit{last})\end{aligned}$$

$$\begin{aligned}&\mathrm{bdll}(L) := \mathrm{dll}(L.\mathit{nodes}, 0, L.\mathit{first}, L.\mathit{last}, 0) \star\\ &(\ L.\mathit{free} < 0 \star \mathrm{range}(L.\mathit{nodes}, -\mathit{free}, L.\mathit{capacity})\\ &\mid L.\mathit{free} > 0 \star \mathrm{sll}(L.\mathit{nodes}, \mathit{free}, 0)\ )\end{aligned}$$

High-level models

$$\text{sequence}\,\langle a \rangle := \text{Nil} \mid \text{Cons } \textbf{of } a * \text{sequence}\,\langle a \rangle$$

$$\text{prod}\,\langle a, b \rangle := \text{Pair } \textbf{of } a * b$$
$$\text{map}\,\langle a, b \rangle := \text{sequence}\,\langle \mathit{prod}\,\langle a, b \rangle \rangle$$

Precise models

$$
\begin{aligned}
&\mathrm{dll}(\mathit{Nodes}, \mathit{first}, \mathit{next}, \mathit{prev}, \mathit{last}) := \\
&\mid \quad \mathit{first} = \mathit{prev} \star \mathit{next} = \mathit{last} \\
&\mid \exists n, X \ . \\
&\qquad \mathit{Nodes} + \mathit{next} \mapsto \{\mathit{Prev} = \mathit{first}, \mathit{Elem} = X, \mathit{Next} = n\} \\
&\qquad \star\, \mathrm{dll}(\mathit{Nodes}, \mathit{next}, n, \mathit{prev}, \mathit{last})
\end{aligned}
$$

Precise models

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

$$\begin{aligned}
&\text{dll}(\mathit{Nodes}, \mathit{first}, \mathit{next}, \mathit{prev}, \mathit{last}, m) := \\
&\mid \quad \mathit{first} = \mathit{prev} \star \mathit{next} = \mathit{last} \star m = C_0 \\
&\mid \exists n, X, m'. \\
&\qquad \mathit{Nodes} + \mathit{next} \mapsto \{\mathit{Prev} = \mathit{first}, \mathit{Elem} = X, \mathit{Next} = n\} \\
&\qquad \star \text{dll}(\mathit{Nodes}, \mathit{next}, n, \mathit{prev}, \mathit{last}, m') \\
&\qquad \star m = C_1
\end{aligned}$$

Precise models

$$\text{precise\_model}\ \langle a \rangle \ := C_0 \mid C_1\ \textbf{of}\ \text{int} * a * \text{precise\_model}\ \langle a \rangle$$

$$
\begin{aligned}
&\text{dll}(Nodes, first, next, prev, last, m) := \\
&\mid \quad first = prev \star next = last \star m = C_0 \\
&\mid \exists n, X, m'. \\
&\qquad Nodes + next \mapsto \{Prev = first, Elem = X, Next = n\} \\
&\qquad \star\, \text{dll}(Nodes, next, n, prev, last, m') \\
&\qquad \star\, m = C_1(n, X, m')
\end{aligned}
$$

Precise models

$$\text{precise\_model}\ \langle a \rangle\ := C_0 \mid C_1\ \textbf{of}\ \text{int} * a * \text{precise\_model}\ \langle a \rangle$$

$$\begin{aligned}
&\text{dll}(\mathit{Nodes}, \mathit{first}, \mathit{next}, \mathit{prev}, \mathit{last}, m) := \textbf{match}\ m\ \textbf{with} \\
&\mid C_0 \rightarrow \mathit{first} = \mathit{prev} \star \mathit{next} = \mathit{last} \\
&\mid C_1(n, X, m') \rightarrow \\
&\qquad \mathit{Nodes} + \mathit{next} \mapsto \{\mathit{Prev} = \mathit{first}, \mathit{Elem} = X, \mathit{Next} = n\} \\
&\qquad \star\, \text{dll}(\mathit{Nodes}, \mathit{next}, n, \mathit{prev}, \mathit{last}, m')
\end{aligned}$$

Precise models

$$\text{precise\_model}\,\langle a\rangle := C_0 \mid C_1\ \textbf{of}\ \text{int} * a * \text{precise\_model}\,\langle a\rangle$$

$$\begin{array}{l}\text{model}(m) := \textbf{match}\ m\ \textbf{with}\\ \mid C_0 \rightarrow \text{Nil}\\ \mid C_1(n, X, m') \rightarrow \text{Cons}(X, \text{model}(m'))\end{array}$$

$$\begin{array}{l}\text{positions}(m, \mathit{first}, i) := \textbf{match}\ m\ \textbf{with}\\ \mid C_0 \rightarrow \text{Nil}\\ \mid C_1(n, X, m') \rightarrow \text{Cons}(\text{Pair}(\mathit{first}, i), \text{positions}(m', n, i+1))\end{array}$$

Precise model composition

$$\text{precise\_model}\,\langle a\rangle := C_0 \mid C_1 \ \textbf{of}\ \text{int} * a * \text{precise\_model}\,\langle a\rangle$$

$$\begin{aligned}&\text{precise\_append}(m_1, m_2) := \textbf{match}\ m_1\ \textbf{with}\\ &\mid C_0 \to m_2\\ &\mid C_1(n, X, m') \to C_1(n, X, \text{precise\_append}(m', m_2))\end{aligned}$$

$$\begin{aligned}&\text{dll}(Nodes, first, next, a, b, m_1) \star \text{dll}(Nodes, a, b, prev, last, m_2) \vdash\\ &\quad \text{dll}(Nodes, first, next, prev, last, \text{precise\_append}(m_1, m_2))\end{aligned}$$

$$\begin{aligned}&\text{dll}(Nodes, first, next, prev, last, \text{precise\_append}(m_1, m_2)) \vdash\\ &\quad \exists a, b.\ \text{dll}(Nodes, first, next, a, b, m_1) \star \text{dll}(Nodes, a, b, prev, last, m_2)\end{aligned}$$

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