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 ($\sim$ 1400 loc)
  - Tested SPARK specification ($\sim$ 3600 loc)
  - Used to verify safety-critical software
- Done:
  - Translation in C ($\sim$ 500 loc)
  - Specification in VeriFast ($\sim$ 1220 loc)
  - Verification in VeriFast ($\sim$ 3500 loc)

Outline

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

Interface 1/2

`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
```

Interface 2/2

`Cursor` is the type of positions inside lists.

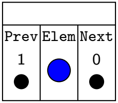
```
No_Element : Cursor
First(List) : Cursor            Last(List) : Cursor
Next(List, Cursor)              Previous(List, Cursor)
Element(List, Cursor) : Element_Type
Find(List, Element_Type, Cursor) : Cursor

Replace_Element(List, Cursor, Element_Type)
Insert(List, Cursor, Element_Type)
Delete(List, Cursor)
...
```

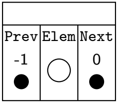
Implementation: Nodes

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.





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.

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.

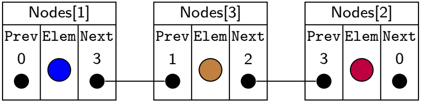
Valid cursors are indexes of occupied nodes.

`No_Element := 0`

Implementation: Lists

Implicit Invariants:

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

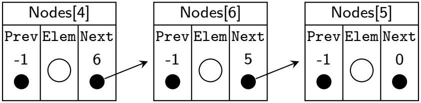


- If the list is initialized, then free nodes form a simply-linked list from `Free` to 0.
- Otherwise, free nodes are the nodes `Nodes[-Free]`, `Nodes[-Free+1]`, …, `Nodes[Capacity]`.

Implementation: Lists

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.

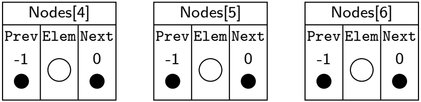


- Otherwise, free nodes are the nodes `Nodes[-Free]`, `Nodes[-Free+1]`, …, `Nodes[Capacity]`.

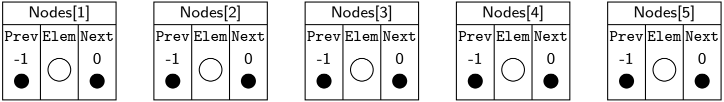
Implementation: Lists

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]`.



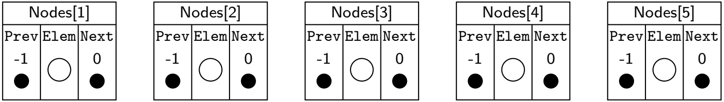
Example



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

```
L = Empty_List(5)
```

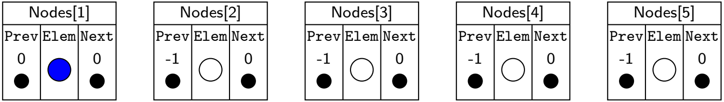
Example



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

`Insert(L, Last(L), e1)`

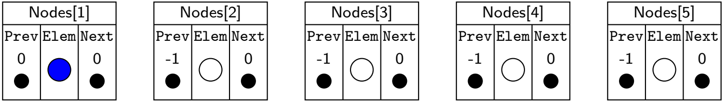
Example



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

`Insert(L, Last(L), e1)`

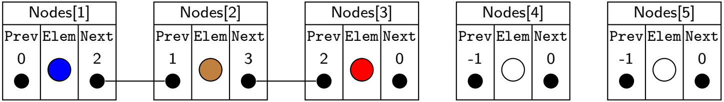
Example



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

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

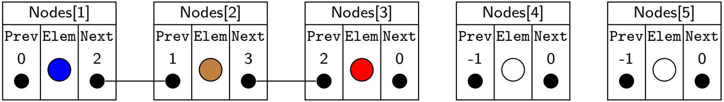
Example



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

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

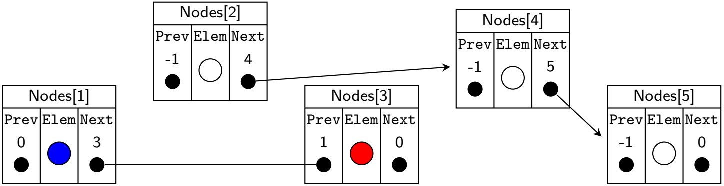
Example



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

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

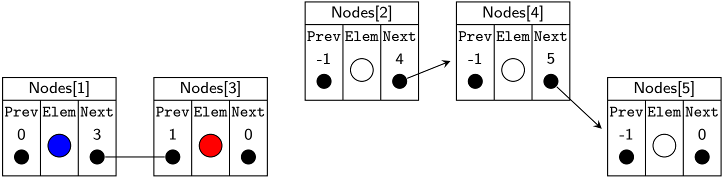
Example



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

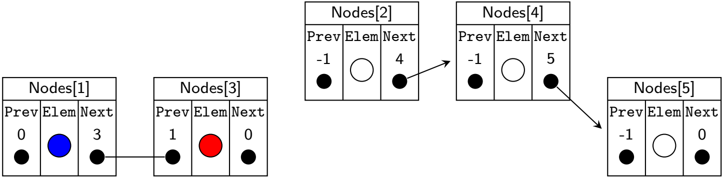
```
c = First(L)
  Next(c)
Delete(L, c)
```

Example



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

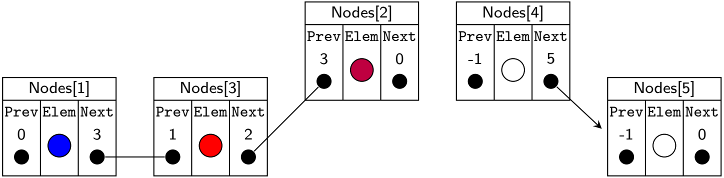
Example



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

`Insert(L, Last(L), e4)`

Example



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

`Insert(L, Last(L), e4)`

Models

Each method of the library is specified by its impact on `Model` and `Positions`.

```
Model(List) : Sequence
Positions(List) : Map(Cursor, Positive)
```

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);
```

Outline

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

VeriFast

VeriFast:

- Verification tool for C (and Java)
- Specification language: separation logic with data types and inductive predicates
- Heuristics
- 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`
- 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:

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

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** $a \ast$ sequence $\langle a \rangle$
  Functions defined by structural recursion
- Inductive predicates
  Example:
  linked_list(x, y) := $(\mathrm{x} = \mathrm{y}) \mid \exists z.\ x \mapsto z \star$ linked_list(z, y)

Low-level invariants

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

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

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

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

High-level models

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

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

High-level models

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

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

$+$ many unusual relations

Precise models

$$
\begin{array}{l}
\mathrm{dll}(\mathit{Nodes}, \mathit{prev}, \mathit{from}, \mathit{last}, \mathit{to} \quad) := \\
\mid \qquad \mathit{prev} = \mathit{last} \star \mathit{from} = \mathit{to} \\
\mid \exists n, X \qquad . \\
\qquad \mathit{Nodes} + \mathit{from} \mapsto \{\mathit{Prev} = \mathit{prev}, \mathit{Elem} = X, \mathit{Next} = n\} \\
\qquad \star\, \mathrm{dll}(\mathit{Nodes}, \mathit{from}, n, \mathit{last}, \mathit{to} \quad)
\end{array}
$$

Precise models

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

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

Precise models

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

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

Precise models

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

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

Precise models

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

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

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

Precise models

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

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

- 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`