Verifying an imperative data structure and its amortized cost analysis using CFML

Arthur Charguéraud

Inria

2014/07/07

Overview of the approach

CFML: characteristic formulae for ML, with Separation Logic.

Describe the semantics of a piece of Caml code as a Coq formula.

Overview of the approach

CFML: characteristic formulae for ML, with Separation Logic.

Describe the semantics of a piece of Caml code as a Coq formula.

```
(** Code.ml **)

let push x s =
  ...
```

```
(** Code_ml.v **)

Axiom push : func.

Axiom push_cf : ∀ x s H Q,
  (...) → App push x s H Q.
```

```
(** Code_proof.v **)

Theorem push_spec : ∀ x s,
  App push x s (...) (...).
Proof.
  apply push_cf.
  ...
Qed.
```

Overview of the approach

CFML: characteristic formulae for ML, with Separation Logic.

Describe the semantics of a piece of Caml code as a Coq formula.

```
(** Code.ml **)

let push x s =
  ...
```

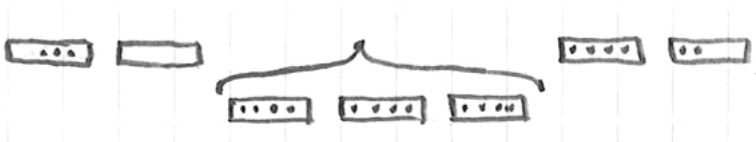
```
(** Code_ml.v **)

Axiom push : func.

Axiom push_cf : ∀ x s H Q,
  (...) → App push x s H Q.
```

```
(** Code_proof.v **)

Theorem push_spec : ∀ x s,
  App push x s (...) (...).
Proof.
  apply push_cf.
  ...
Qed.
```

Characteristic formulae: sound, complete, compositional, of linear size.

A case study

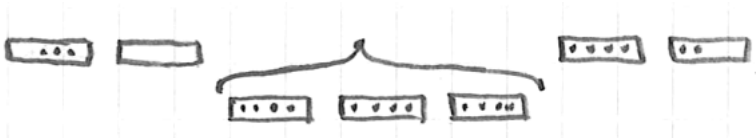
Chunked sequences



Efficient push and pop operations at front and back

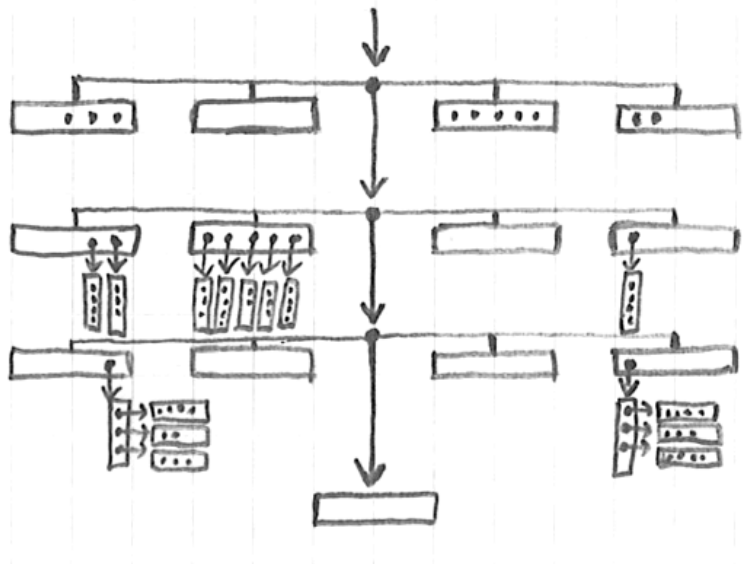
A case study

Chunked sequences



Efficient push and pop operations at front and back

Bootstrapped chunked sequences



Same, plus log-time concatenation and split

Contents

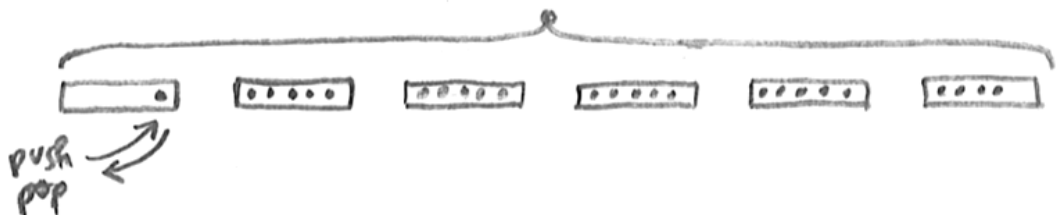
Problem with sequences of chunks

A sequence of chunks, based on ring buffers of fixed capacity K:

Problem with sequences of chunks

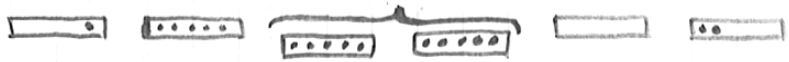
A sequence of chunks, based on ring buffers of fixed capacity K:

Problem with series of consecutive push-pop operations:

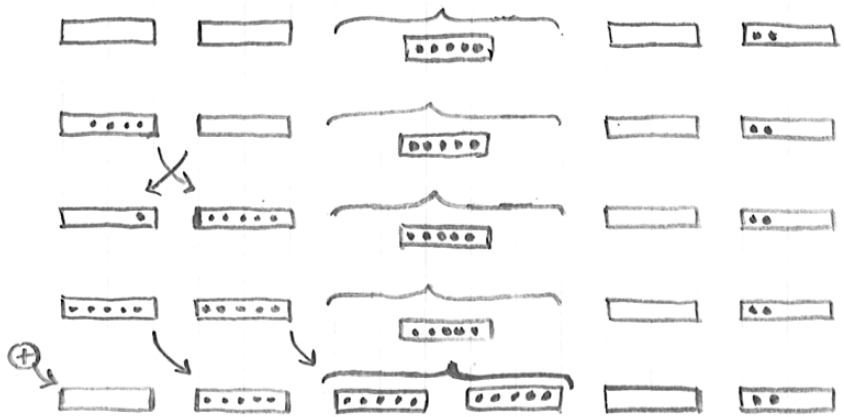


A solution to the problem

Keep two special chunks on each side.

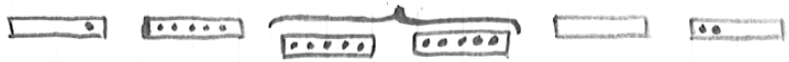


Push a chunk into the middle only when both special chunks are full.

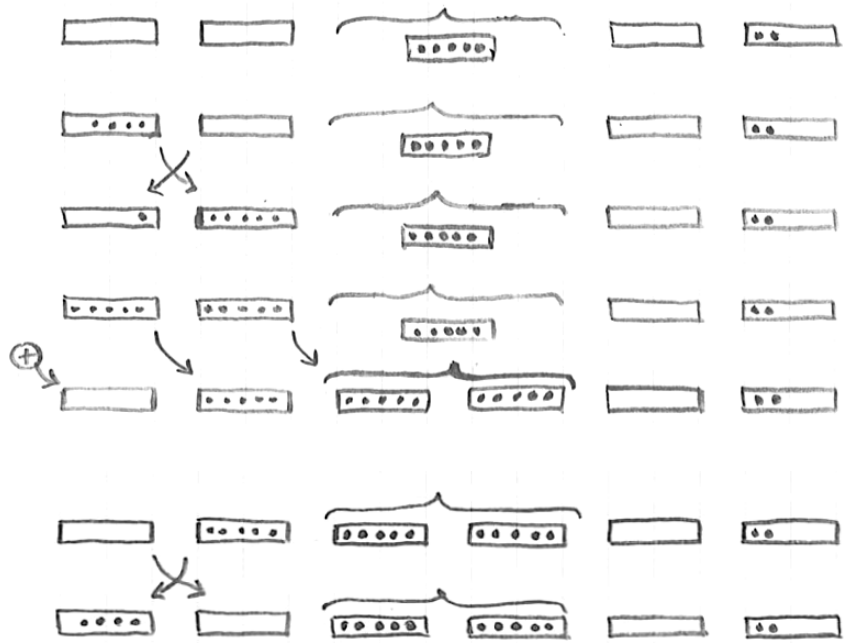


A solution to the problem

Keep two special chunks on each side.

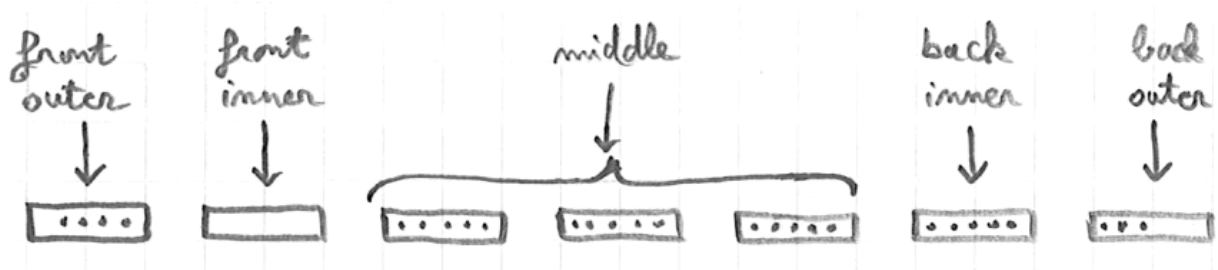


Push a chunk into the middle only when both special chunks are full.



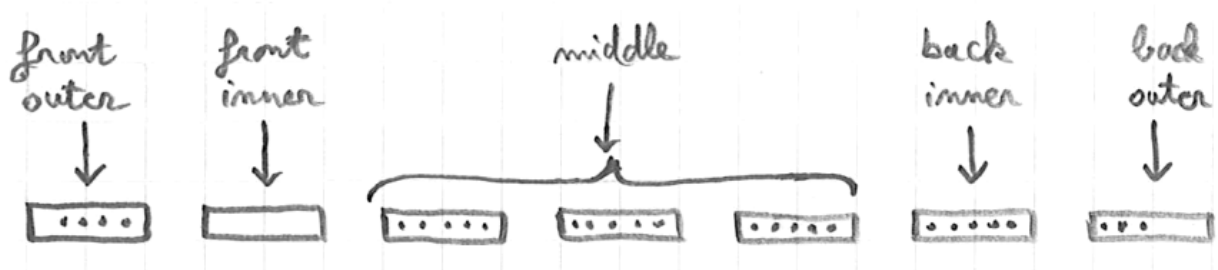
Every allocation is amortized over at least K push operations.

Implementation of chunked sequences




```
module Make (Chunk : SequenceSig.FixedCapacityS) (Middle : SequenceSig.S) =
struct
      type 'a t = {
        mutable front_outer : 'a Chunk.t;
        mutable front_inner : 'a Chunk.t;
        mutable middle : ('a Chunk.t) Middle.t;
        mutable back_inner : 'a Chunk.t;
        mutable back_outer : 'a Chunk.t; }
      ...
end
```

Implementation of chunked sequences

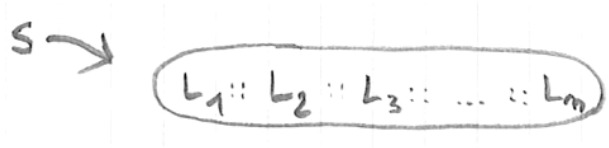



```
module Make (Chunk : SequenceSig.FixedCapacityS) (Middle : SequenceSig.S) =
struct
       type 'a t = {
          mutable front_outer : 'a Chunk.t;
          mutable front_inner : 'a Chunk.t;
          mutable middle : ('a Chunk.t) Middle.t;
          mutable back_inner : 'a Chunk.t;
          mutable back_outer : 'a Chunk.t; }
       ...
end
```

Invariant: front-inner and back-inner chunks are either empty or full, and the middle sequence contains only nonempty chunks.

Specification of sequence operations

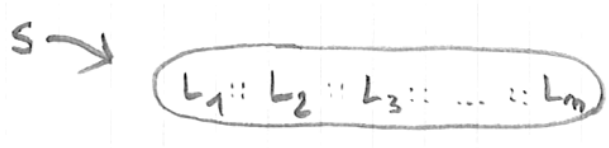
$$(s \rightsquigarrow \mathsf{Seq}\, L) \;:\; \mathsf{Heap} \to \mathsf{Prop}$$



$\forall (A : \mathsf{Type})\ (L : \mathsf{list} A)\ (x : A)\ (s : \mathsf{loc}).$
$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda tt.\ s \rightsquigarrow \mathsf{Seq}\, (x :: L))$

Specification of sequence operations

$(s \rightsquigarrow \mathsf{Seq}\, L)$: Heap → Prop

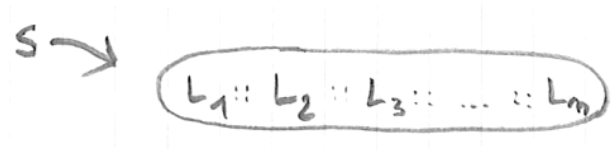


$\forall (A : \mathsf{Type})\ (L : \mathsf{list} A)\ (x : A)\ (s : \mathsf{loc}).$
App push_front $x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda tt.\ s \rightsquigarrow \mathsf{Seq}\, (x :: L))$

App append $s_1\ s_2\ (s_1 \rightsquigarrow \mathsf{Seq}\, L_1\ *\ s_2 \rightsquigarrow \mathsf{Seq}\, L_2)\ (\lambda tt.\ s_1 \rightsquigarrow \mathsf{Seq}\, (L_1 +\!\!+ L_2)))$

Specification of sequence operations

$$(s \rightsquigarrow \mathsf{Seq}\, L) \;:\; \mathsf{Heap} \to \mathsf{Prop}$$



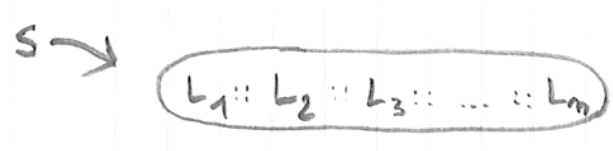
$\forall (A : \mathsf{Type})\ (L : \mathsf{list} A)\ (x : A)\ (s : \mathsf{loc}).$
$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda tt.\ s \rightsquigarrow \mathsf{Seq}\, (x :: L))$

$\mathsf{App}\ \mathsf{append}\ s_1\ s_2\ (s_1 \rightsquigarrow \mathsf{Seq}\, L_1\ *\ s_2 \rightsquigarrow \mathsf{Seq}\, L_2)\ (\lambda tt.\ s_1 \rightsquigarrow \mathsf{Seq}\, (L_1 +\!\!+ L_2)))$

$L \neq \mathsf{nil} \Rightarrow$
$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda x.\, \exists L'.\ [L = x :: L'] * (s \rightsquigarrow \mathsf{Seq}\, L'))$

Specification of sequence operations

$(s \rightsquigarrow \mathsf{Seq}\, L)$: Heap → Prop



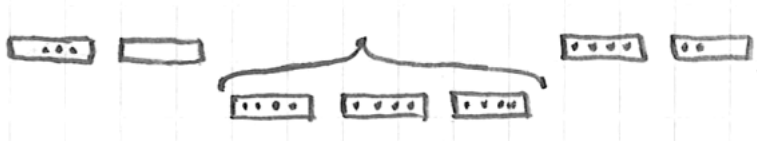
$\forall (A : \mathsf{Type})\ (L : \mathsf{list} A)\ (x : A)\ (s : \mathsf{loc}).$
App push_front $x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda tt.\ s \rightsquigarrow \mathsf{Seq}\, (x :: L))$

App append $s_1\ s_2\ (s_1 \rightsquigarrow \mathsf{Seq}\, L_1\ *\ s_2 \rightsquigarrow \mathsf{Seq}\, L_2)\ (\lambda tt.\ s_1 \rightsquigarrow \mathsf{Seq}\, (L_1 +\!\!+ L_2)))$

$L \neq \mathsf{nil} \Rightarrow$
App pop_front $s\ (s \rightsquigarrow \mathsf{Seq}\, L)\ (\lambda x.\, \exists L'.\ [L = x :: L'] * (s \rightsquigarrow \mathsf{Seq}\, L'))$

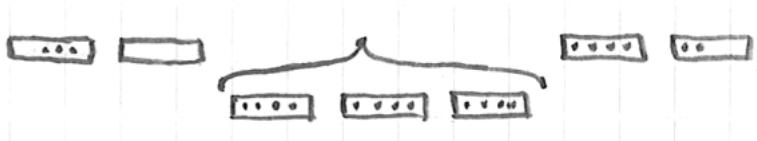
App_1 :
$\forall AB.$ func → A → (Heap → Prop) → (B → Heap → Prop) → Prop

Representation predicates

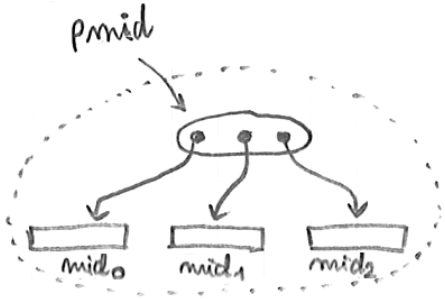


ChunkedSeq:	$s \rightsquigarrow \mathsf{Seq}\, L$	where	$L : \mathsf{list}\, A$
Chunk:	$s \rightsquigarrow \mathsf{chunk}\, L$	where	$L : \mathsf{list}\, A$
Middle:	$s \rightsquigarrow \mathsf{middle}\, L$	where	$L : \mathsf{list}\, \mathsf{loc}$

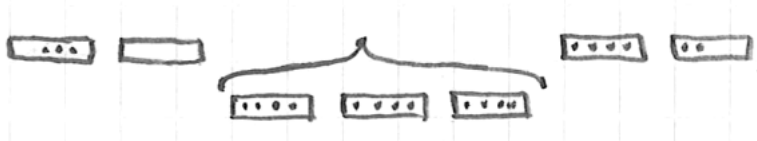
Representation predicates



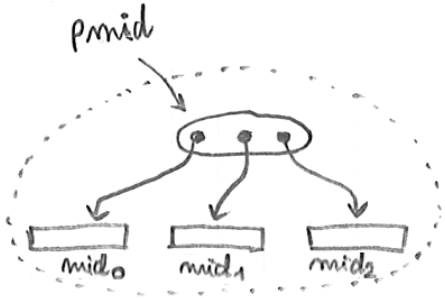
ChunkedSeq:	$s \rightsquigarrow \mathsf{Seq}\, L$	where	$L : \mathsf{list}\, A$
Chunk:	$s \rightsquigarrow \mathsf{chunk}\, L$	where	$L : \mathsf{list}\, A$
Middle:	$s \rightsquigarrow \mathsf{middle}\, L$	where	$L : \mathsf{list}\, \mathsf{loc}$
	$s \rightsquigarrow (\mathsf{middleof}\, \mathsf{chunk})\, L$	where	$L : \mathsf{list}\, (\mathsf{list}\, A)$

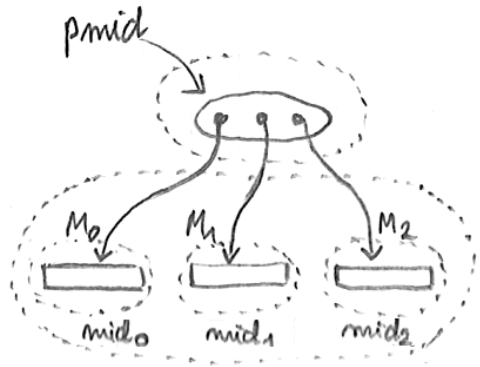


Representation predicates



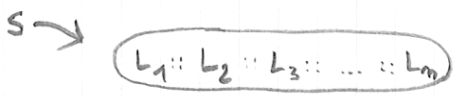
ChunkedSeq:	$s \rightsquigarrow \mathsf{Seq}\, L$	where $L : \mathsf{list}\, A$
Chunk:	$s \rightsquigarrow \mathsf{chunk}\, L$	where $L : \mathsf{list}\, A$
Middle:	$s \rightsquigarrow \mathsf{middle}\, L$	where $L : \mathsf{list\,loc}$
	$s \rightsquigarrow (\mathsf{middleof\,chunk})\, L$	where $L : \mathsf{list}\,(\mathsf{list}\, A)$



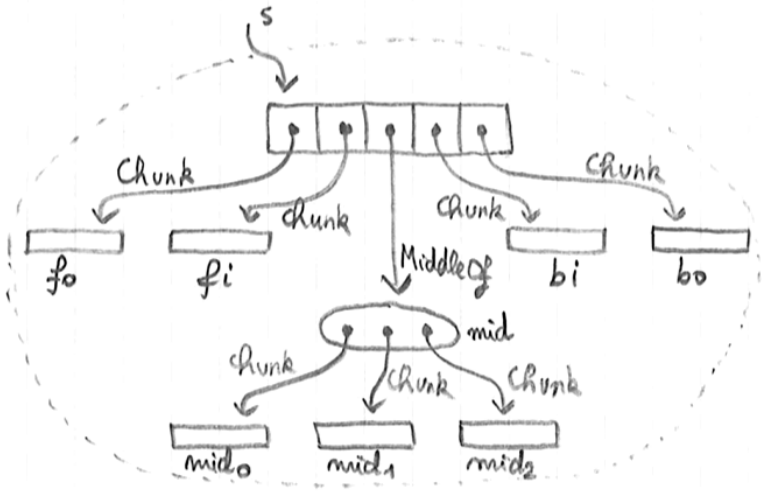


$$p \rightsquigarrow (\mathsf{middleof\,chunk})\, L \;=\; \exists M.\; p \rightsquigarrow \mathsf{middle}\, M \;*\; \Big(\circledast_{i}\; \mathsf{M}_i \rightsquigarrow \mathsf{chunk}\, L_i\Big)$$

Definition of Seq



$$(s \rightsquigarrow \mathsf{Seq}\, L) \;\equiv\; (\mathsf{Seq}\, L\, s)$$



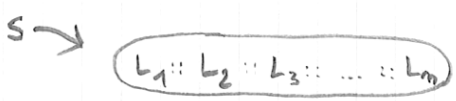
Definition Seq $(A : \mathsf{Type})$ $(L : \mathsf{list}\, A)$ $(s : \mathsf{loc})$: Heap → Prop :=

∃fo fi mid bi bo.

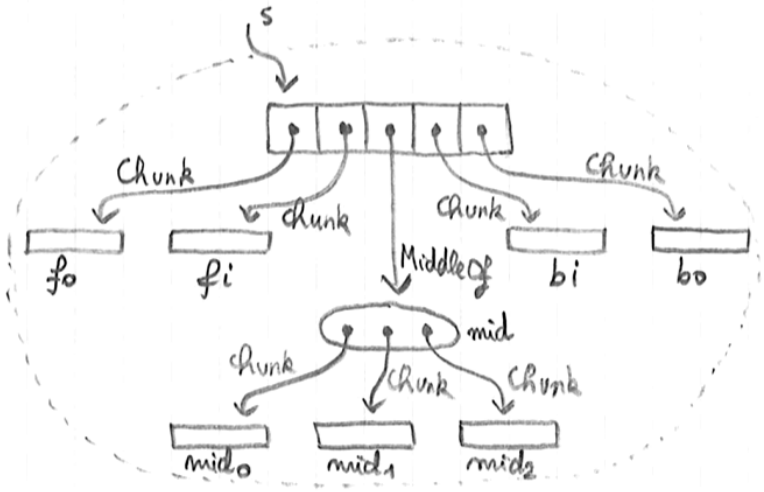
∗ $s \rightsquigarrow$ Tof chunk chunk (middleof chunk) chunk chunk fo fi mid bi bo

∗ [inv L fo fi mid bi bo].

Definition of Seq



$$(s \rightsquigarrow \mathsf{Seq}\, L) \;\equiv\; (\mathsf{Seq}\, L\, s)$$



Definition Seq $(A : \mathsf{Type})$ $(L : \mathsf{list}\, A)$ $(s : \mathsf{loc})$: Heap → Prop :=

$\exists$fo fi mid bi bo.

$*$ $s \rightsquigarrow$ Tof chunk chunk (middleof chunk) chunk chunk fo fi mid bi bo

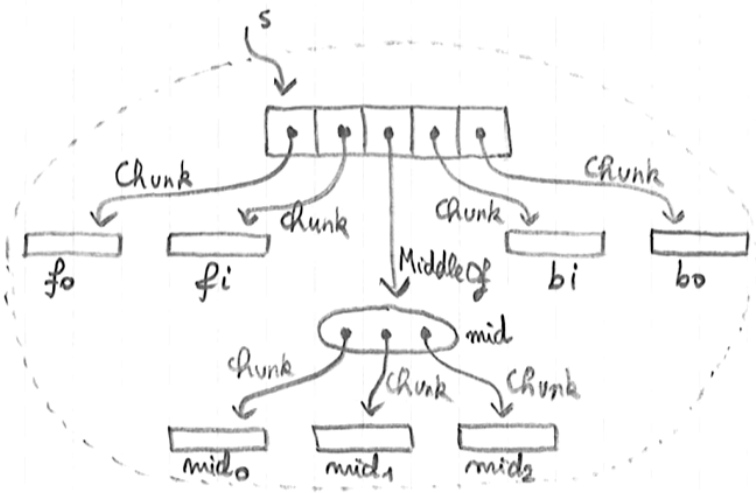
$*$ [inv L fo fi mid bi bo].

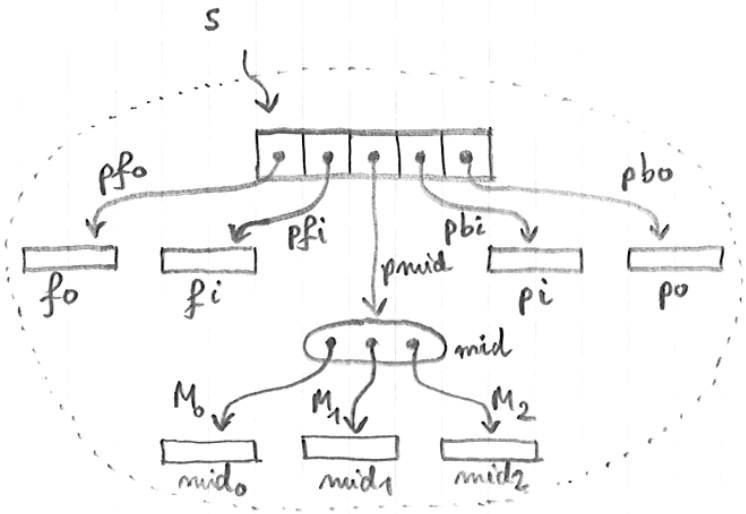
Definition inv $(A : \mathsf{Type})$ (fo fi : list A) (mid : list(list A)) (bi bo : list A) :=

$L =$ fo ++ fi ++ concat mid ++ bi ++ bo

$\wedge$ (fi = nil $\vee$ $|\mathsf{fi}| = K$) $\wedge$ (bi = nil $\vee$ $|\mathsf{bi}| = K$) $\wedge$ (Forall ($\neq$ nil) mid).

Recursive ownership





$$(s \rightsquigarrow \text{Tof chunk chunk (middleof chunk) chunk chunk fo fi mid bi bo}) =$$

$$\exists\ \text{pfo pfi pmid pbi pbo}.\ s \rightsquigarrow \text{record\_t pfo pfi pmid pbi pbo}$$

$$* \ \text{pfo} \rightsquigarrow \text{chunk fo} \ * \ \text{pfi} \rightsquigarrow \text{chunk fi}$$

$$* \ \text{pbi} \rightsquigarrow \text{chunk bi} \ * \ \text{pbo} \rightsquigarrow \text{chunk bo}$$

$$* \ \text{pmid} \rightsquigarrow (\text{middleof chunk})\ \text{mid}$$

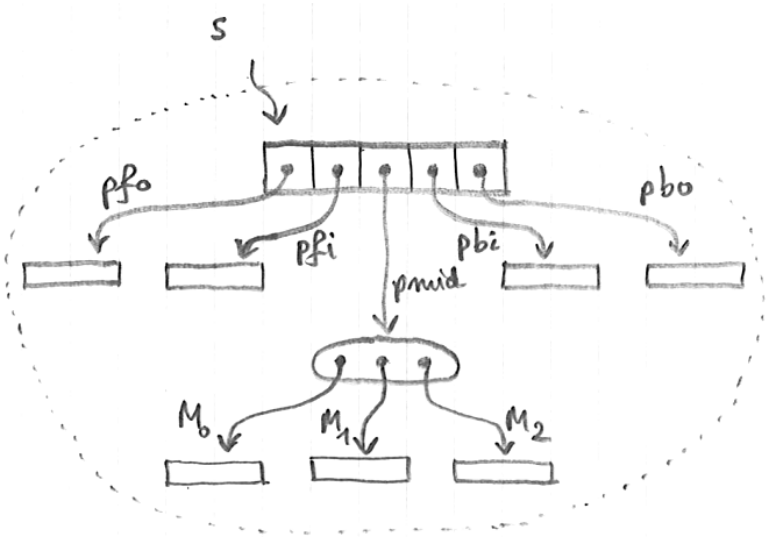
Recursive ownership

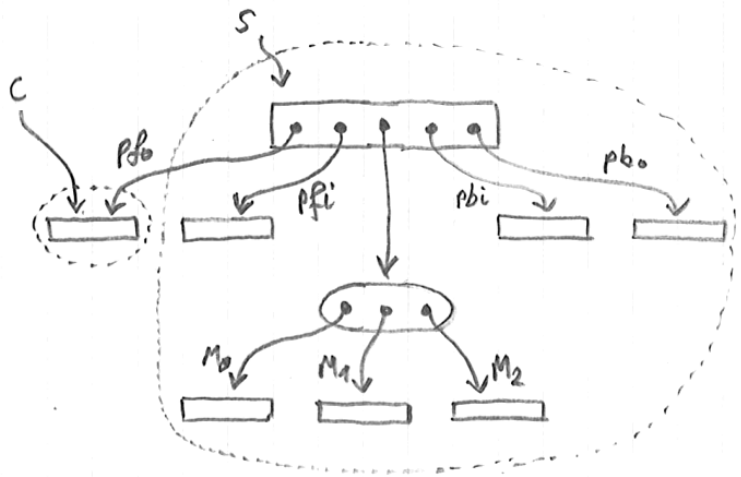
$$
\begin{aligned}
&(s \rightsquigarrow \mathsf{Tof}\, R_1\, R_2\, R_3\, R_4\, R_5\, X_1\, X_2\, X_3\, X_4\, X_5) \;\equiv \\
&\quad \exists x_1\, x_2\, x_3\, x_4\, x_5.\; s \rightsquigarrow \mathsf{record\_t}\, x_1\, x_2\, x_3\, x_4\, x_5 \\
&\quad * (x_1 \rightsquigarrow R_1\, X_1) * (x_2 \rightsquigarrow R_2\, X_2) \\
&\quad * (x_3 \rightsquigarrow R_3\, X_3) * (x_4 \rightsquigarrow R_4\, X_4) * (x_5 \rightsquigarrow R_5\, X_5)
\end{aligned}
$$

$$
\begin{aligned}
&(s \rightsquigarrow \mathsf{Tof\ chunk\ chunk\ (middleof\ chunk)\ chunk\ chunk\ \ fo\ fi\ mid\ bi\ bo}) \;= \\
&\quad \exists\ \mathsf{pfo\ pfi\ pmid\ pbi\ pbo}.\; s \rightsquigarrow \mathsf{record\_t\ pfo\ pfi\ pmid\ pbi\ pbo} \\
&* \ \mathsf{pfo} \rightsquigarrow \mathsf{chunk\ fo} \ * \ \mathsf{pfi} \rightsquigarrow \mathsf{chunk\ fi} \\
&* \ \mathsf{pbi} \rightsquigarrow \mathsf{chunk\ bi} \ * \ \mathsf{pbo} \rightsquigarrow \mathsf{chunk\ bo} \\
&* \ \mathsf{pmid} \rightsquigarrow \mathsf{(middleof\ chunk)\ mid}
\end{aligned}
$$

Control of recursive ownership

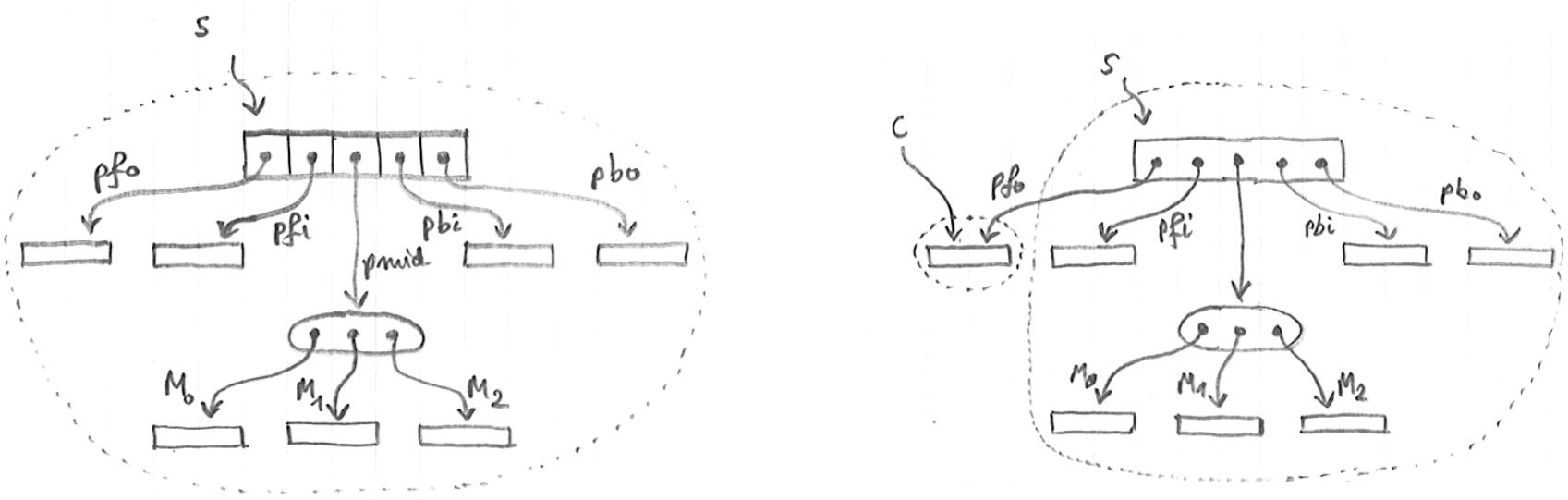
```
let c = s.front_outer in Chunk.push_front x c
```





Control of recursive ownership

```
let c = s.front_outer in Chunk.push_front x c
```

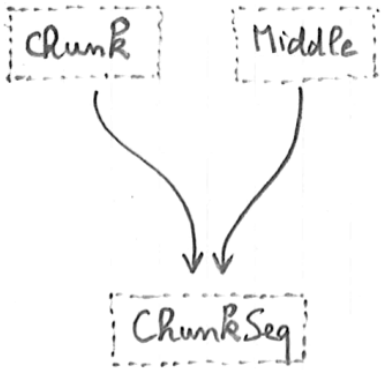


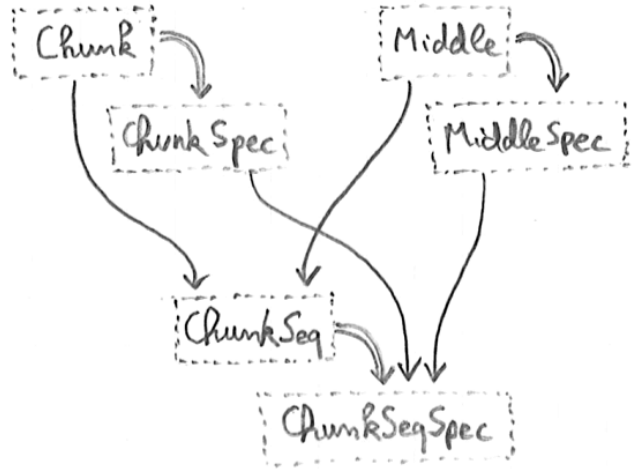
$$s \rightsquigarrow \mathsf{T\,of\,chunk\,chunk\,(middle\,of\,chunk)\,chunk\,chunk\,fo\,fi\,mid\,bi\,bo}$$

$$= \quad \exists \mathsf{pfo}.\quad s \rightsquigarrow \mathsf{T\,of\,Id\,chunk\,(middle\,of\,chunk)\,chunk\,chunk\,pfo\,fi\,mid\,bi\,bo}$$
$$\quad * \ \mathsf{pfo} \rightsquigarrow \mathsf{chunk\,fo}$$

where Id is such that: $(\mathsf{x} \rightsquigarrow \mathsf{Id\,X}) \equiv [x = X]$.

Modular verification



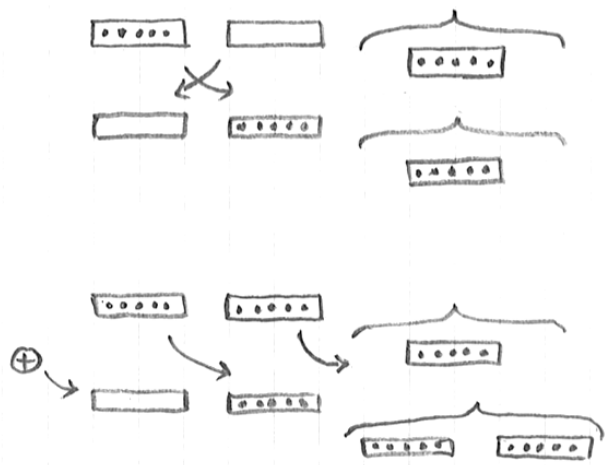



```
(** Code_ml.v **)

Module ChunkedSeq
  (Chunk : ...)
  (Middle : ...).

Axiom push : func.

Axiom push_cf : ...
```

```
(** Code_proof.v **)

Module ChunkedSeqSpec
   (Chunk : ...)
   (ChunkSpec : SeqFixedSpec with Module Q := Chunk)
   (Middle : ...)
   (MiddleSpec : SeqSpec with Module Q := Middle).

Module Import Q := ChunkedSeq Chunk Middle.

Theorem push_spec : ... App Q.push x s ...
```

Source code of push-front

```
let push_front x s =
   let co = s.front_outer in
   if Chunk.is_full co then begin
      let ci = s.front_inner in
      s.front_inner <- co;
      if Chunk.is_empty ci then begin
         s.front_outer <- ci;
      end else begin
         Middle.push_front ci s.middle;
         s.front_outer <- Chunk.create();
      end
   end;
   Chunk.push_front x s.front_outer
```



Automatically-generated characteristic formula

```
Axiom push_front : func.

Axiom push_front_cf :
  (@CFPrint.tag tag_top_fun _ _ (@CFPrint.tag tag_body
  _ _ (∀ _A : Type, (∀ K : (_A → ((t _A) →
  ((CFHeaps.hprop → ((unit → CFHeaps.hprop) → Prop))
  → Prop))), ((is_spec_2 K) → ((∀ x : _A, (∀
  s : (t _A), (((K x) s) (@CFPrint.tag tag_let_trm (Label_create
  'co) _ (local (fun H : CFHeaps.hprop ⇒ (fun Q : (_
  → CFHeaps.hprop) ⇒ (Logic.ex (fun Q1 : ((chunk _A)
  → CFHeaps.hprop) ⇒ ((Logic.and (((@CFPrint.tag tag_apply
  _ _ ((((@app_1 (t _A)) (chunk _A)) _get_front_outer)
  s)) H) Q1)) (∀ co : (chunk _A), (((@CFPrint.tag
  tag_let_trm (Label_create '_x1) _ (local (fun H : CFHeaps.hprop
  ⇒ (fun Q : (_ → CFHeaps.hprop) ⇒ (Logic.ex (fun ...
  (* ..... goes on for 60 more lines ..... *)
```

Verification of push-front

```
Lemma push_front_spec : ∀ (A:Type) (L:list A) (x:A) (q:Q.t A),
  App push_front x q (q ⤳ Seq L) (fun (_:unit) ⇒ q ⤳ Seq (x::L)).
Proof.
  xcf. intros. hunfold Seq at 1. xextract as fo fi mid bi bo Inv.
  xapp_by focus. xapps. xseq. xif_post (∃ (fo2 fi2 : list A),
    ∃mid2, s ⤳ TF fo2 fi2 mid2 bi bo
    ∗ [inv L fo2 fi2 mid2 bi bo] ∗ [~ full fo2]).
  xapp_by focus. xapp_by unfocus. xapps. xif.
  xapp_by unfocus. hsimpl∗. destruct Inv. constructor~.
  xapp_by focus. destruct Inv as [MI MF MB MN MC].
  destruct MF; tryfalse. xapp~. xapp. xapp_by unfocus.
  hchange (_unfocus_middle s). hsimpl∗. constructor∗.
    applys∗ middle_conseq_push_full.
  xret. hchange (_unfocus_front_outer s). hsimpl∗.
  xok. clears_all Inv. xextract as fo fi mid Inv Nfo.
  xapp_by focus. xapp. hchange (_unfocus_front_outer s).
  hunfold Seq. hsimpl. destruct Inv. constructor∗.
Qed.
```

Interactive proof

```
MI : L = fo ++ fi ++ concat mid ++ bi ++ bo
H : full fi
MB : bi = nil ∨full bi
MN : middle_chunks_ok mid
C : LibList.length fo = ChunkSpec.Q.capacity
C0 : fi ≠nil
------------------------------------------------------
 ((App Middle.push_front ci _x12 ;) ;;
  (Let _x15 := App Chunk.create tt ; in
   App _set_front_outer s _x15 ;))
 (ci ⇝ RChunk fi ∗
    s ⇝ Tof Id RChunk Id RChunk RChunk co fo _x12 bi bo ∗
    _x12 ⇝ (RMiddleOf RChunk) mid)
 (fun _ ⇒ ∃fo2 fi2 mid2,
    s ⇝ TofAll fo2 fi2 mid2 bi bo ∗
    [inv L fo2 fi2 mid2 bi bo] ∗ [∼ full fo2])
```

Amortized analysis using time credits

Time credits:

$$\$\,x \quad : \quad \text{Heap} \to \text{Prop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;*\; \$\,y \quad \text{and} \quad \$\,0 \;=\; [\,]$$

Amortized analysis using time credits

Time credits:

$$\$\,x \quad : \quad \text{Heap} \to \text{Prop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;*\; \$\,y \quad \text{and} \quad \$\,0 \;=\; [\,]$$

Principle:

The execution of any atomic operation costs $\$1$.

Amortized analysis using time credits

Time credits:

$$\$\,x \quad : \quad \text{Heap} \to \text{Prop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) \;=\; \$\,x \;*\; \$\,y \quad \text{and} \quad \$\,0 \;=\; [\,]$$

Principle:

The execution of any atomic operation costs $\$1$.

Simplification:

Calling a non-inlinable function or entering a loop body costs $\$1$.

Time credits in pre-conditions

App Chunk.create v $(\$\, C_{\mathsf{chunkcreate}})(\lambda c.\ c \rightsquigarrow \mathsf{chunk}\,\mathsf{nil})$

App Chunk.push_front x s $(s \rightsquigarrow \mathsf{chunk}\, L * \$\, C_{\mathsf{chunkop}})$ $(...)$

App Middle.push_front x s $(... * \$\, C_{\mathsf{middleop}})$ $(...)$

Time credits in pre-conditions

App Chunk.create v $(\$\, C_{\mathsf{chunkcreate}})(\lambda c.\ c \rightsquigarrow \mathsf{chunk}\,\mathsf{nil})$

App Chunk.push_front x s $(s \rightsquigarrow \mathsf{chunk}\, L * \$\, C_{\mathsf{chunkop}})$ $(...)$

App Middle.push_front x s $(... * \$\, C_{\mathsf{middleop}})$ $(...)$

App Chunk.pop_front x s $(... * \$\, 0)$ $(...)$

App Middle.push_front x s $(... * \$\, 0)$ $(...)$

Amortized cost of pop operations

Naive approach: account for the cost of both push and pop operations.

$$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$\, C_{\mathsf{push}})\ (...)$$
$$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}\, (x :: L)\ *\ \$\, C_{\mathsf{pop}})\ (...)$$

Amortized cost of pop operations

Naive approach: account for the cost of both push and pop operations.

$$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}\ L\ *\ \$\,C_{\mathsf{push}})\ (\ldots)$$
$$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}\ (x :: L)\ *\ \$\,C_{\mathsf{pop}})\ (\ldots)$$

Idea: charge each pop operation to the corresponding push operation.

$$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}'\ L\ *\ \$(C_{\mathsf{push}} + C_{\mathsf{pop}}))\ (\ldots)$$
$$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}'\ (x :: L)\ *\ \$\,0)\ (\ldots)$$

Amortized cost of pop operations

Naive approach: account for the cost of both push and pop operations.

$$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$\, C_{\mathsf{push}})\ (...)$$
$$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}\, (x :: L)\ *\ \$\, C_{\mathsf{pop}})\ (...)$$

Idea: charge each pop operation to the corresponding push operation.

$$\mathsf{App}\ \mathsf{push\_front}\ x\ s\ (s \rightsquigarrow \mathsf{Seq}'\, L\ *\ \$(C_{\mathsf{push}} + C_{\mathsf{pop}}))\ (...)$$
$$\mathsf{App}\ \mathsf{pop\_front}\ s\ (s \rightsquigarrow \mathsf{Seq}'\, (x :: L)\ *\ \$\, 0)\ (...)$$

where:

$$(s \rightsquigarrow \mathsf{Seq}'\, L) \ \equiv\ (s \rightsquigarrow \mathsf{Seq}\, L)\ *\ \$(|L| \cdot C_{\mathsf{pop}})$$

Analysis of push-front using time credits

App Chunk.create v ($\$\, C_{\mathsf{chunkcreate}}$)($\lambda c.\ c \rightsquigarrow \mathsf{chunk}\,\mathsf{nil}$)

App Chunk.push_front $x\ s$ ($s \rightsquigarrow \mathsf{chunk}\, L\ *\ \$\, C_{\mathsf{chunkop}}$) (...)

App Middle.push_front $x\ s$ (... $*\ \$\, C_{\mathsf{middleop}}$) (...)

App ChunkedSeq.push_front $x\ s$ ($s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$(C_{\mathsf{chunkop}} + ...)$) (...)

Analysis of push-front using time credits

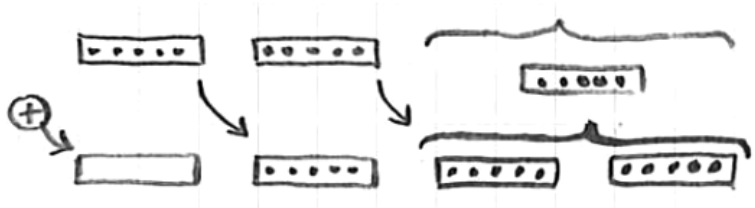
App Chunk.create v ($\$\, C_{\mathsf{chunkcreate}}$)($\lambda c.\ c \rightsquigarrow \mathsf{chunk}\,\mathsf{nil}$)

App Chunk.push_front $x\ s$ ($s \rightsquigarrow \mathsf{chunk}\, L\ *\ \$\, C_{\mathsf{chunkop}}$) (...)

App Middle.push_front $x\ s$ (... $*\ \$\, C_{\mathsf{middleop}}$) (...)

App ChunkedSeq.push_front $x\ s$ ($s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$(C_{\mathsf{chunkop}} + ...)$) (...)

Expansive transition, paid for at most every K push-front operations:



$\$(C_{\mathsf{chunkcreate}} + C_{\mathsf{middleop}})$

Analysis of push-front using time credits

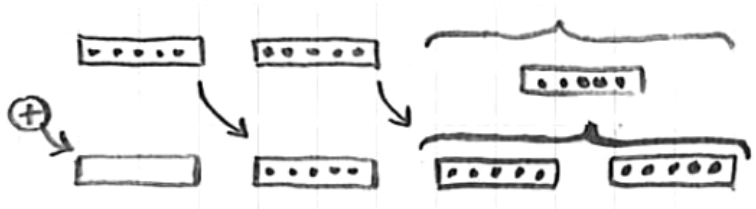
App Chunk.create v ($\$\, C_{\mathsf{chunkcreate}}$)($\lambda c.\ c \rightsquigarrow \mathsf{chunk}\,\mathsf{nil}$)

App Chunk.push_front $x\ s$ ($s \rightsquigarrow \mathsf{chunk}\, L\ *\ \$\, C_{\mathsf{chunkop}}$) (...)

App Middle.push_front $x\ s$ (... $*\ \$\, C_{\mathsf{middleop}}$) (...)

App ChunkedSeq.push_front $x\ s$ ($s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$(C_{\mathsf{chunkop}} + ...)$) (...)

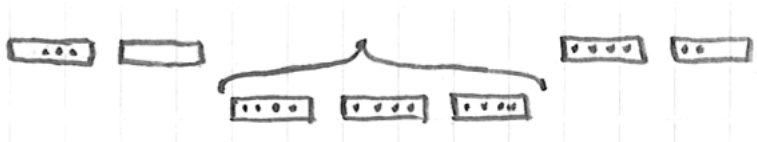
Expansive transition, paid for at most every K push-front operations:



$\$(C_{\mathsf{chunkcreate}} + C_{\mathsf{middleop}})$

App push_front $x\ s$ ($s \rightsquigarrow \mathsf{Seq}\, L\ *\ \$(C_{\mathsf{chunkop}} + \frac{C_{\mathsf{chunkcreate}} + C_{\mathsf{middleop}}}{K})$) (...)

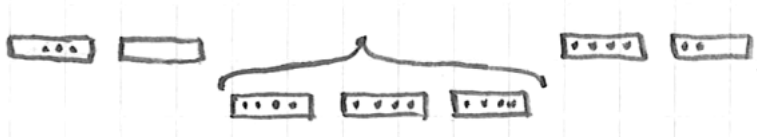
Definition of the potential



$$(s \rightsquigarrow \mathsf{Seq}\ L) \quad \equiv \quad \exists \mathsf{fo\ fi\ mid\ bi\ bo}.\ s \rightsquigarrow \mathsf{T\,fo\ fi\ mid\ bi\ bo}$$
$$* \ [\mathsf{inv}\ L\ \mathsf{fo\ fi\ mid\ bi\ bo}]$$
$$* \ \$(\mathsf{side\_potential\ fo\ fi} + \mathsf{side\_potential\ bo\ bi}).$$

$$\mathsf{side\_potential\ co\ ci} \quad \equiv \quad \mathsf{if}\ (\mathsf{ci} = \mathsf{nil})\ \mathsf{then}\ 0\ \mathsf{else}\ (|\mathsf{co}| \cdot \frac{C_{\mathsf{chunkcreate}} + C_{\mathsf{middle}}}{K})$$

Definition of the potential



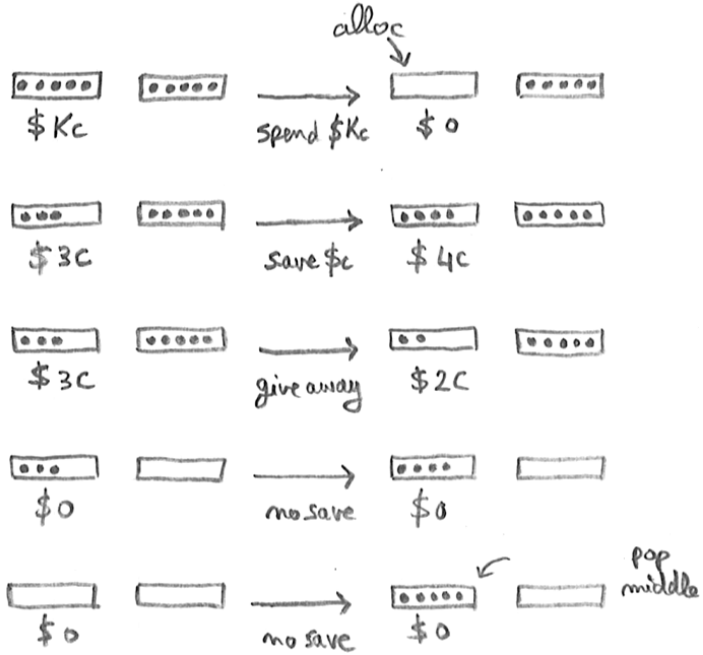
$$(s \rightsquigarrow \mathsf{Seq}\ L) \quad \equiv \quad \exists \mathsf{fo\ fi\ mid\ bi\ bo}.\ s \rightsquigarrow \mathsf{T\,fo\ fi\ mid\ bi\ bo} * [\mathsf{inv}\ L\ \mathsf{fo\ fi\ mid\ bi\ bo}] * \$(\mathsf{side\_potential\ fo\ fi} + \mathsf{side\_potential\ bo\ bi}).$$

$$\mathsf{side\_potential\ co\ ci} \quad \equiv \quad \mathsf{if}\ (\mathsf{ci} = \mathsf{nil})\ \mathsf{then}\ 0\ \mathsf{else}\ (|\mathsf{co}| \cdot \tfrac{C_{\mathsf{chunkcreate}} + C_{\mathsf{middle}}}{K})$$

Remark: $K \cdot \frac{C_{\mathsf{chunkcreate}} + C_{\mathsf{middle}}}{K}$ matches the cost of the expansive transition.

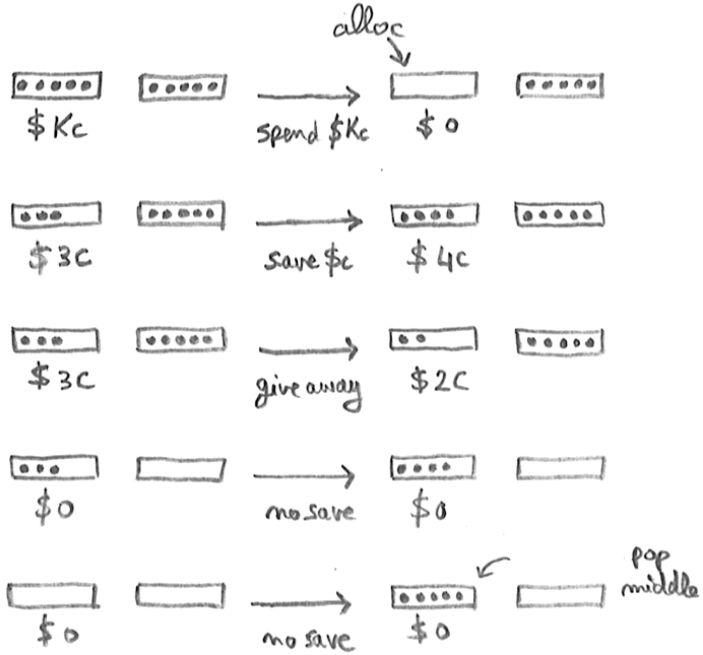
Proof obligations associated with potential

$$\text{side\_potential fo fi} + \frac{C_{\text{chunkcreate}} + C_{\text{middleop}}}{K} \geqslant \text{side\_potential fo' fi'}$$



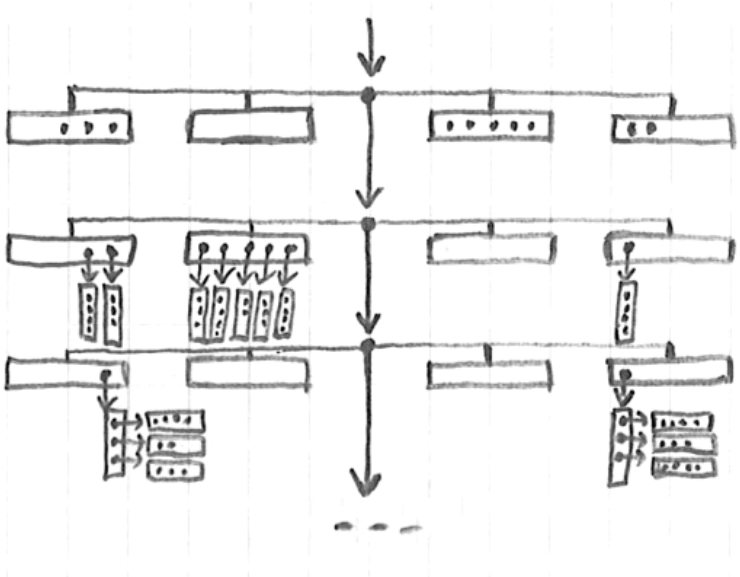
Proof obligations associated with potential

$$\text{side\_potential fo fi} + \frac{C_{\text{chunkcreate}} + C_{\text{middleop}}}{K} \geqslant \text{side\_potential fo' fi'}$$



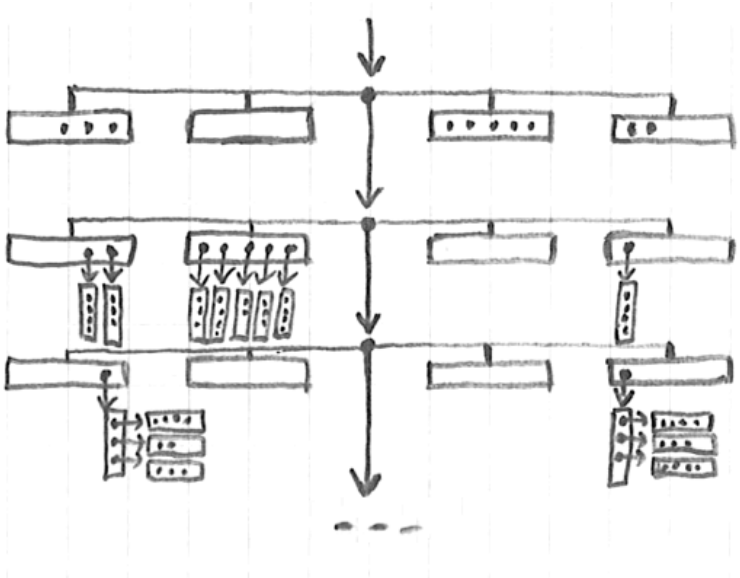

```
Ltac pot_simpl :=
  unfold side_potential;
  simpl_credits_ineq;
  repeat case_if;
  normalize_list_length;
  normalize_credits_arith.
```

Bootstrapped chunked sequence



```
type 'a t = {
  mutable front_outer : 'a Chunk.t;
  mutable front_inner : 'a Chunk.t;
  mutable middle : ('a Chunk.t) t;
  mutable back_inner : 'a Chunk.t;
  mutable back_outer : 'a Chunk.t; }
```

Bootstrapped chunked sequence

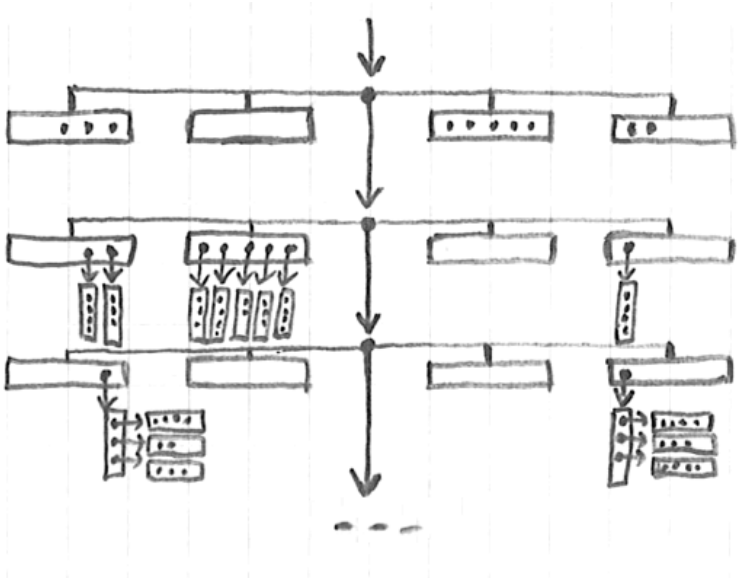


```
type 'a t = {
  mutable front_outer : 'a Chunk.t;
  mutable front_inner : 'a Chunk.t;
  mutable middle : ('a Chunk.t) t;
  mutable back_inner : 'a Chunk.t;
  mutable back_outer : 'a Chunk.t; }
```

Cost of push:

$$\approx \; C + \tfrac{1}{K} \cdot (C + \tfrac{1}{K} \cdot (C + \tfrac{1}{K} \cdot (C + ...))) \;=\; C \cdot \tfrac{K}{K-1} \;\approx\; C$$

Bootstrapped chunked sequence



```
type 'a t = {
  mutable front_outer : 'a Chunk.t;
  mutable front_inner : 'a Chunk.t;
  mutable middle : ('a Chunk.t) t;
  mutable back_inner : 'a Chunk.t;
  mutable back_outer : 'a Chunk.t; }
```

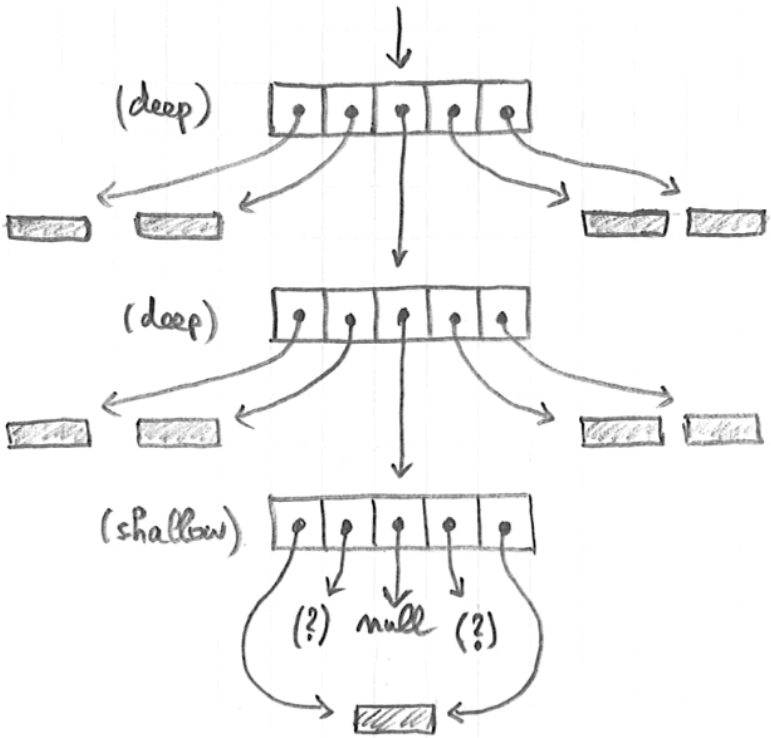
Cost of push:

$$\approx \; C + \tfrac{1}{K} \cdot (C + \tfrac{1}{K} \cdot (C + \tfrac{1}{K} \cdot (C + ...))) \;=\; C \cdot \tfrac{K}{K-1} \;\approx\; C$$

Cost of append:

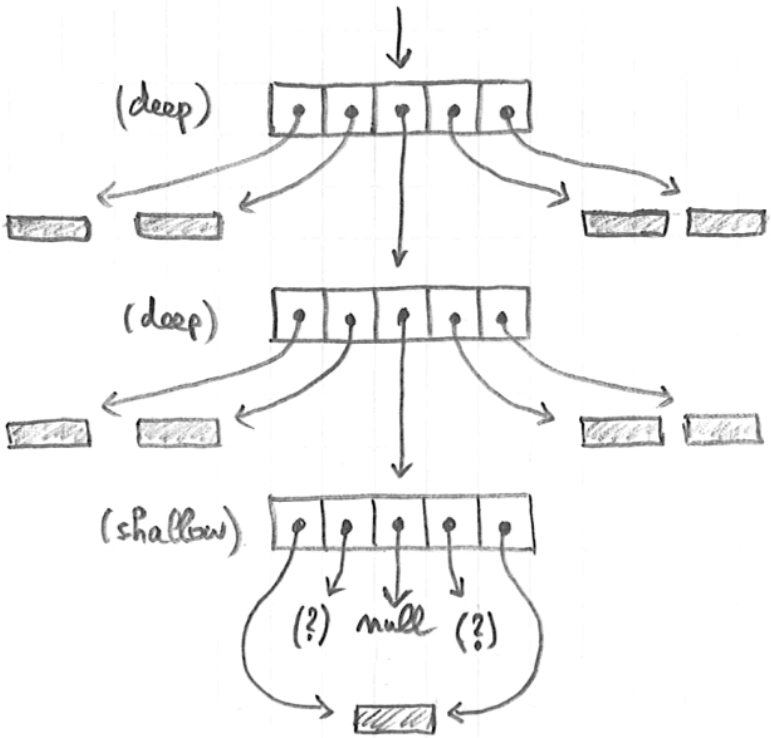
$$\approx \; C' \times K \times (1 + \lfloor \log_{(K+1)/2}(n) \rfloor)$$

Implementation of the bootstrap




```
type 'a t = {
  mutable front_outer : 'a Chunk.t;
  mutable front_inner : 'a Chunk.t;
  mutable middle : ('a Chunk.t) t;
  mutable back_inner : 'a Chunk.t;
  mutable back_outer : 'a Chunk.t; }
```

Implementation of the bootstrap




```
type 'a t = {
  mutable front_outer : 'a Chunk.t;
  mutable front_inner : 'a Chunk.t;
  mutable middle : ('a Chunk.t) t;
  mutable back_inner : 'a Chunk.t;
  mutable back_outer : 'a Chunk.t; }
```

```
let rec push_front : 'a. 'a -> 'a t -> unit = fun x s ->
   if not (Chunk.is_full s.front_outer) then
      Chunk.push_front x s.front_outer
   else
      ...
```

Specification in terms of pure trees

```
Inductive tree (A : Type) : Type :=
  | shallow : list A → tree A
  | deep : list A → list A → tree (list A) → list A → list A → tree A.
```

Representation predicate:

$$(s \rightsquigarrow \mathsf{Seqof}\, R\, L) \equiv \exists T.\ (s \rightsquigarrow \mathsf{Treeof}\, R\, T) * [\mathsf{tinv}\, L\, T]$$

Specialization for the top layer: $(s \rightsquigarrow \mathsf{Seq}\, L) \equiv (s \rightsquigarrow \mathsf{Seqof}\, \mathsf{Id}\, L)$.

Specification in terms of pure trees

```
Inductive tree (A : Type) : Type :=
  | shallow : list A → tree A
  | deep : list A → list A → tree (list A) → list A → list A → tree A.
```

Representation predicate:

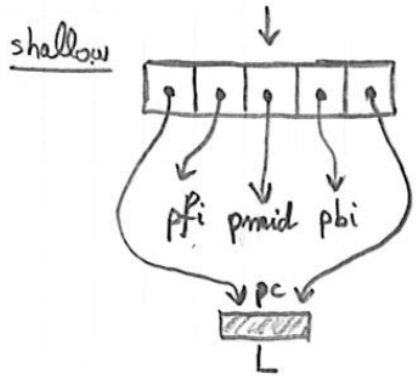
$$(s \rightsquigarrow \mathsf{Seqof}\, R\, L) \equiv \exists T.\ (s \rightsquigarrow \mathsf{Treeof}\, R\, T) * [\mathsf{tinv}\, L\, T]$$

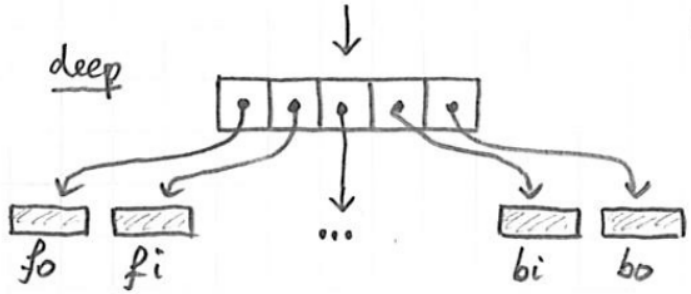
Specialization for the top layer: $(s \rightsquigarrow \mathsf{Seq}\, L) \equiv (s \rightsquigarrow \mathsf{Seqof}\, \mathsf{Id}\, L)$.

Definition of the invariant on trees:

$$\frac{|L| > 1 \qquad \mathsf{inv}\, L\, \mathsf{fo}\, \mathsf{fi}\, \mathsf{mid}\, \mathsf{bi}\, \mathsf{bo} \qquad \mathsf{tinv}\, \mathsf{mid}\, \mathsf{tmid}}{\mathsf{tinv}\, L\, (\mathsf{deep}\, \mathsf{fo}\, \mathsf{fi}\, \mathsf{tmid}\, \mathsf{bi}\, \mathsf{bo})} \qquad \frac{}{\mathsf{tinv}\, L\, (\mathsf{shallow}\, L)}$$

Recursive representation predicate





$(s \rightsquigarrow \mathsf{Treeof\ R}\ T)\ \equiv$

match T with

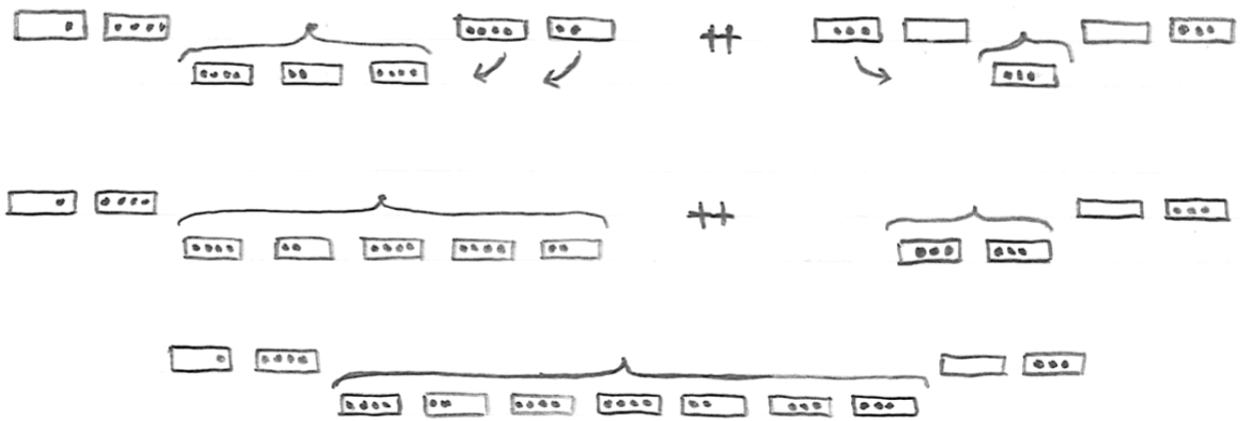
| shallow L $\Rightarrow$ $\exists$pc pfi pmid pbi.
  $s \rightsquigarrow$ Tof Id Id Id Id Id pc pfi pmid pbi pc $*$ pc $\rightsquigarrow$ chunk L

| deep fo fi tmid bi bo $\Rightarrow$
  $s \rightsquigarrow$ Tof cR cR (Treeof cR) cR cR fo fi tmid bi bo
  $*$ \$(side_potential fo fi + side_potential bo bi).

where cR $\equiv$ chunkof R.

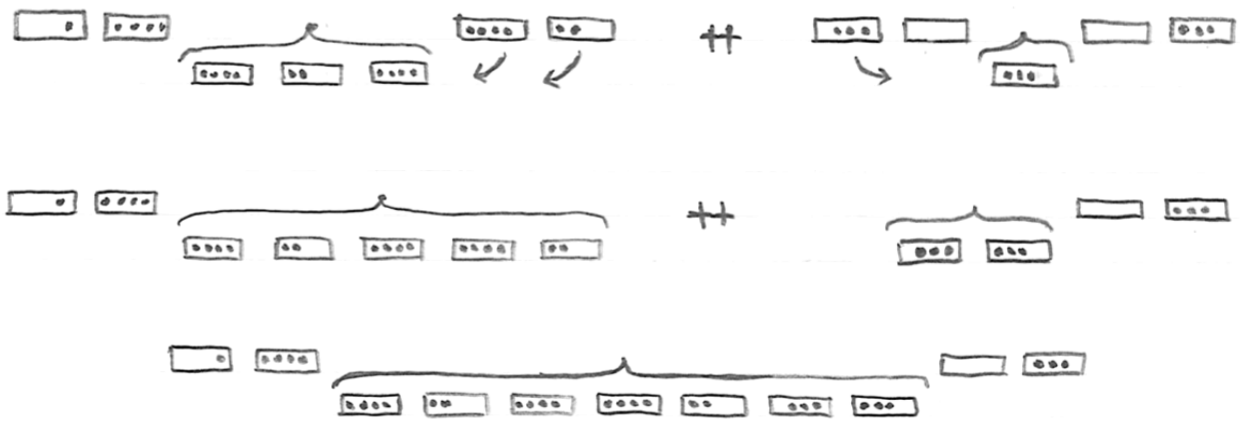
Append on chunked sequences

Implementation:



Append on chunked sequences

Implementation:

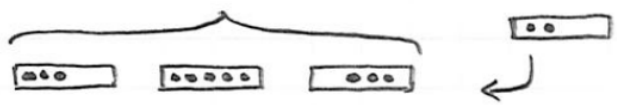


Definition inv $(A : \mathsf{Type})$ $(\mathsf{fo}\,\mathsf{fi} : \mathsf{list}A)$ $(\mathsf{mid} : \mathsf{list}(\mathsf{list}A))$ $(\mathsf{bi}\,\mathsf{bo} : \mathsf{list}A) :=$

$\ldots \quad \wedge \quad \mathsf{ForallConseq}\,(\lambda c_1 c_2.\ |c_1| + |c_2| > K)\,\mathsf{mid}.$

Auxiliary function for append

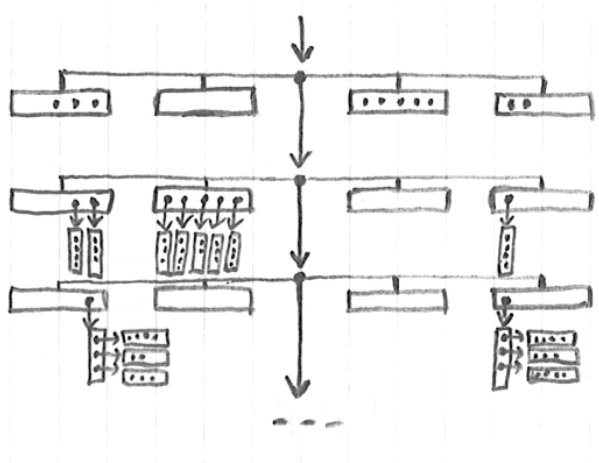
```
let middle_merge_back m c =
   let sc = Chunk.length c in
   if sc > 0 then begin
      if is_empty m then begin
         push_back c m
      end else begin
         let b = pop_back m in
         let sb = Chunk.length b in
         if sc + sb > capacity then begin
            push_back b m;
            push_back c m
         end else begin
            Chunk.append b c;
            push_back b m;
         end
      end
   end
```



Implementation of append

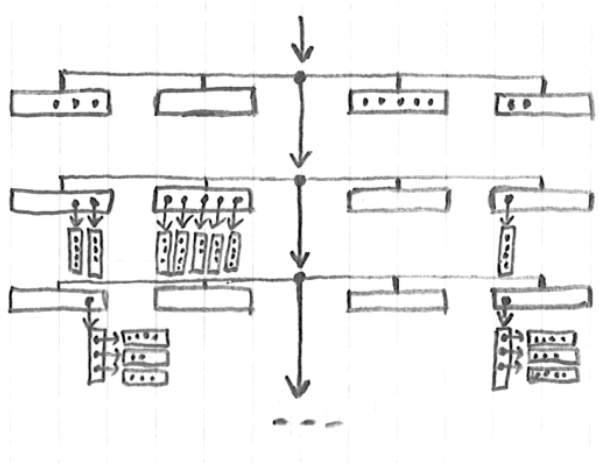
```
let rec append : 'a. 'a t -> 'a t -> unit =
fun s1 s2 ->
  if is_shallow s2 then begin
    append_chunk_back s2.front_outer s1
  end else if is_shallow s1 then begin
    swap_shallow_with_other s1 s2;
    append_chunk_front s2.front_outer s1
  end else begin
    let m1 = s1.middle in
    let ci = s1.back_inner in
    let co = s1.back_outer in
    if Chunk.is_empty ci then begin
      middle_merge_back m1 co
    end else begin
      push_back ci m1;
      if not (Chunk.is_empty co)
        then push_back co m1;
    end;
    let m2 = s2.middle in
    let ci = s2.front_inner in
    let co = s2.front_outer in
    if Chunk.is_empty ci then begin
      middle_merge_front m2 co
    end else begin
      push_front ci m2;
      if not (Chunk.is_empty co)
        then push_front co m2;
    end;
    s1.back_inner <- s2.back_inner;
    s1.back_outer <- s2.back_outer;
    if not (is_empty m1)
    && not (is_empty m2) then begin
      let c1 = pop_back m1 in
      let sc1 = Chunk.length c1 in
      let c2 = pop_front m2 in
      let sc2 = Chunk.length c2 in
      if sc1 + sc2 > Capacity then
      begin
        push_back c1 m1;
        push_front c2 m2;
      end else begin
        Chunk.append c1 c2;
        push_back c1 m1;
      end;
    end;
    append m1 m2;
  end
```

Cost analysis for append



Cost of append $\approx O(K) \times \log_{(K+1)/2}(\min(|L_1|, |L_2|))$.

Cost analysis for append



Cost of append $\approx O(K) \times \log_{(K+1)/2}(\min(|L_1|, |L_2|))$.

App append $s_1\ s_2$

$$\left(s_1 \rightsquigarrow \mathsf{Seq}\, L_1 \ * \ s_2 \rightsquigarrow \mathsf{Seq}\, L_2 \ * \ \$\, cK(1 + \min(\mathsf{D}(L_1), \mathsf{D}(L_2)))\right)(\ldots)$$

where:

$$\mathsf{D}(L) \quad \equiv \quad \text{if } |L| \leqslant 1 \text{ then } 0 \text{ else } 1 + \log_{(K+1)/2}(|L| - 1)$$

Related work

Separation Logic

- O'Hearn, Reynolds, Yang (2001-2002)

Separation Logic in Coq

- Nanevski, Morrisett, Birkedal, Chlipala et al (Ynot, 2005-2009)

Amortised Resource Analysis With Separation Logic

- Atkey (2010-2011)

Most general Hoare-triples

- Honda, Berger, Yoshida (2004-2007)

Future work

- Improved error messages
- More efficient tactics
- Better proof automation
- “Big-O” analysis

Pointers

CFML

- Characteristic Formulae for the Verification of Imperative Programs (ICFP 2011 and HOSC 2014)
- Examples, and course notes:

  `http://arthur.chargueraud.org/softs/cfml`

Boostrapped chunked sequences

- Theory and Practice of Chunked Sequences (U. Acar, A. Charguéraud and M. Rainey, ESA 2014)