Light-weight containers for Isabelle: efficient, extensible, nestable

Andreas Lochbihler

Institute of Information Security
ETH Zurich

ITP 2013

From formal specifications to implementations

specification HOL

$$f\ x = \textit{if}\ \exists y.\ \dots\ \textit{then}\ \iota A.\ \dots\ \textit{else}\ \{\}$$

SML implementation

```
fun f x = compute (init x)
fun init x = ...
fun compute A = ... compute ...
```

From formal specifications to implementations

specification

HOL

$f\ x = if\ \exists y.\ \dots\ then\ \iota A.\ \dots\ else\ \{\}$

prove

$f\ x = compute\ (init\ x)$

$init\ x = \dots$

$compute\ A = \dots\ compute\ \dots$

code generator

```
fun f x = compute (init x)
fun init x = ...
fun compute A = ... compute ...
```

SML

implementation

From formal specifications to implementations

specification

HOL

$f\ x = \mathit{if}\ \exists y.\ \ldots\ \mathit{then}\ \iota A.\ \ldots\ \mathit{else}\ \{\}$

prove

How can we easily use efficient data structures without cluttering proofs?

$f\ x = \mathit{compute}\ (\mathit{init}\ x)$
$\mathit{init}\ x = \ldots$
$\mathit{compute}\ A = \ldots\ \mathit{compute}\ \ldots$

code generator

```
fun f x = compute (init x)
fun init x = ...
fun compute A = ... compute ...
```

SML

implementation

From formal specifications to implementations

specification

HOL

$f\ x = \mathit{if}\ \exists y.\ \ldots\ \mathit{then}\ \iota A.\ \ldots\ \mathit{else}\ \{\}$

prove

How can we easily use efficient data structures without cluttering proofs?

$f\ x = \mathit{compute}\ (\mathit{init}\ x)$

$\mathit{init}\ x = \ldots$

$\mathit{compute}\ A = \ldots\ \mathit{compute}\ \ldots$

SML

implementat

ICF & AutoRef [ITP'10, ITP'12, ITP'13]

- refine to efficient data structures **in HOL**
- uniform interface for container implementations (RBTs, ...)
- tool support for refinement proofs

From formal specifications to implementations

specification $f\ x = if\ \exists y.\ \ldots\ then\ \iota A.\ \ldots\ else\ \{\}$

HOL

prove

How can we easily use

Light-weight containers (LC)

- refine to efficient data structures **in the code generator**
- stick to abstract container types (sets, maps, ...)

ICF & AutoRef [ITP'10, ITP'12, ITP'13]

- refine to efficient data structures **in HOL**
- uniform interface for container implementations (RBTs, ...)
- tool support for refinement proofs

SML

implementat

Example: convert Boolean formulas to CNF

theory *Clauses* imports *Main* begin

datatype *bexp* = *Var nat* | *Not bexp* | *And bexp bexp* | *Or bexp bexp*

type_synonym *literal* = *nat* × *bool*
type_synonym *clause* = *literal set*
type_synonym *cnf* = *clause set*

shallow embedding
of CNF formulas:
set of sets of literals

Example: convert Boolean formulas to CNF

theory *Clauses* imports *Main* begin

datatype $bexp = Var\ nat \mid Not\ bexp \mid And\ bexp\ bexp \mid Or\ bexp\ bexp$

type_synonym $literal = nat \times bool$
type_synonym $clause = literal\ set$
type_synonym $cnf = clause\ set$

shallow embedding of CNF formulas: **set of sets of literals**

function $cnf :: bexp \Rightarrow cnf$ where

$$
\begin{aligned}
& cnf\ (Var\ v) = \{\{(v, True)\}\} \\
\mid\ & cnf\ (And\ b\ b') = cnf\ b \cup cnf\ b' \\
\mid\ & cnf\ (Or\ b\ b') = \bigcup_{c \in cnf\ b} (\lambda c'.\ c \cup c')\ `\ cnf\ b' \\
\mid\ & cnf\ (Not\ (Var\ v)) = \{\{(n, False)\}\} \\
\mid\ & cnf\ (Not\ (Not\ b)) = cnf\ b \\
\mid\ & cnf\ (Not\ (And\ b\ b')) = cnf\ (Or\ (Not\ b)\ (Not\ b')) \\
\mid\ & cnf\ (Not\ (Or\ b\ b')) = cnf\ (And\ (Not\ b)\ (Not\ b'))
\end{aligned}
$$

Example: convert Boolean formulas to CNF

theory *Clauses* imports *Main* begin

datatype *bexp* = *Var nat* | *Not bexp* | *And bexp bexp* | *Or bexp bexp*

type_synonym *literal* = *nat* × *bool*
type_synonym *clause* = *literal set*
type_synonym *cnf* = *clause set*

shallow embedding of CNF formulas: **set of sets of literals**

function *cnf* :: *bexp* ⇒ *cnf* where

$$
\begin{aligned}
& cnf\ (Var\ v) = \{\{(v, True)\}\} \\
| \ & cnf\ (And\ b\ b') = cnf\ b \cup cnf\ b' \\
| \ & cnf\ (Or\ b\ b') = \bigcup_{c \in cnf\ b} (\lambda c'.\ c \cup c')\ `\ cnf\ b' \\
| \ & cnf\ (Not\ (Var\ v)) = \{\{(n, False)\}\} \\
| \ & cnf\ (Not\ (Not\ b)) = cnf\ b \\
| \ & cnf\ (Not\ (And\ b\ b')) = cnf\ (Or\ (Not\ b)\ (Not\ b')) \\
| \ & cnf\ (Not\ (Or\ b\ b')) = cnf\ (And\ (Not\ b)\ (Not\ b'))
\end{aligned}
$$

definition *test* :: *bexp* where *test* = ...

7168 clauses

value [code] *cnf test* = {}

takes 57 s

Example: convert Boolean formulas to CNF

theory *Clauses* imports *Main* **Containers** begin

datatype *bexp* = *Var nat* | *Not bexp* | *And bexp bexp* | *Or bexp bexp*

type_synonym *literal* = *nat* × *bool*
type_synonym *clause* = *literal set*
type_synonym *cnf* = *clause set*

shallow embedding of CNF formulas: **set of sets of literals**

function *cnf* :: *bexp* ⇒ *cnf* where

$cnf\ (Var\ v) = \{\{(v, True)\}\}$
$|\ cnf\ (And\ b\ b') = cnf\ b \cup cnf\ b'$
$|\ cnf\ (Or\ b\ b') = \bigcup_{c \in cnf\ b} (\lambda c'.\ c \cup c')\ `\ cnf\ b'$
$|\ cnf\ (Not\ (Var\ v)) = \{\{(n, False)\}\}$
$|\ cnf\ (Not\ (Not\ b)) = cnf\ b$
$|\ cnf\ (Not\ (And\ b\ b')) = cnf\ (Or\ (Not\ b)\ (Not\ b'))$
$|\ cnf\ (Not\ (Or\ b\ b')) = cnf\ (And\ (Not\ b)\ (Not\ b'))$

definition *test* :: *bexp* where *test* = . . .

7168 clauses

value [code] *cnf test* = {}

takes ~~**57 s**~~ **1.3 s**

Example: convert Boolean formulas to CNF

theo

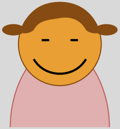
data

type
type
type ... erals

func

defi

Criteria for container frameworks

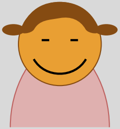


ease of use

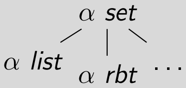
value [code] *cnf test* $= \{\}$ **takes** ~~57 s~~ 1.3 s

Example: convert Boolean formulas to CNF

Criteria for container frameworks



ease of use



flexibility

value [code] *cnf test* = {} **takes** ~~57 s~~ 1.3 s

Example: convert Boolean formulas to CNF

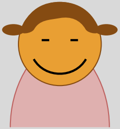
thec

data

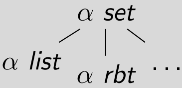
type
type
type

func

Criteria for container frameworks



ease of use



flexibility

$$\frac{\alpha\ \textit{list} \quad \alpha\ \textit{rbt}}{\textit{int set} \quad \textit{string set}}$$

$+\ \alpha$ **trie**

$+\ (\alpha \Rightarrow \beta)$ **set**

extensibility

g

erals

defi

value [code] *cnf test* $= \{\}$ **takes** ~~57 s~~ 1.3 s

Example: convert Boolean formulas to CNF

theo

data

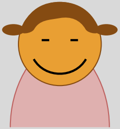
type g

type

type erals

func

Criteria for container frameworks



ease of use

α *set*

α *list* — α *rbt* — . . .

flexibility

α *list*

α *rbt*

$+\ \alpha$ **trie**

int set

string set

$+\ (\alpha \Rightarrow \beta)$ **set**

extensibility

int set

int set set

int set set set

. . . *set set set set*

nestability

defi

value [code] *cnf test* $= \{\}$ **takes** ~~57 s~~ 1.3 s

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

	constructor	**operation** $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set}$	$(x \in \{\, y.\ P\ y \,\}) = P\ x$
impl.		

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

	constructor	**operation** $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	$(x \in \{\, y.\ P\ y \,\}) = P\ x$
impl.	$set, coset :: \alpha\ list \Rightarrow \alpha\ set$	

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

	constructor	operation $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	$(x \in \{y.\ P\ y\}) = P\ x$
impl.	$set, coset :: \alpha\ list \Rightarrow \alpha\ set$	

new pseudo-constructors

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

	constructor	operation $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set}$	$(x \in \{\, y.\ P\ y \,\}) = P\ x$
impl.	$\mathit{set}, \mathit{coset} :: \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{set}$	$(x \in \mathit{set}\ \mathit{xs}) = (\ \mathit{memb}\ (\mathit{op} =)\ \mathit{xs}\ x)$ $(x \in \mathit{coset}\ \mathit{xs}) = (\neg \mathit{memb}\ (\mathit{op} =)\ \mathit{xs}\ x)$

new pseudo-constructors

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

	constructor	**operation** $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	$(x \in \{\, y.\ P\ y\,\}) = P\ x$
impl.	$set, coset :: \alpha\ list \Rightarrow \alpha\ set$	$(x \in set\ xs) = (\ memb\ (op =)\ xs\ x)$ $(x \in coset\ xs) = (\neg memb\ (op =)\ xs\ x)$

new pseudo-constructors

pattern-matching on pseudo-constructors

Refinement in the code generator

Refinement separates code generation from specification.

- logically insignificant
- can be changed and extended at any time
- key to extensibility and modularity

Example: Implement α *set* by lists

depend on overloaded operation

	constructor	**operation** $\in$
spec.	$\{\_.\ \_\} :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$	$(x \in \{y.\ P\ y\}) = P\ x$
impl.	$set, coset :: \alpha\ list \Rightarrow \alpha\ set$	$(x \in set\ xs) = (memb\ (op =)\ xs\ x)$ $(x \in coset\ xs) = (\neg memb\ (op =)\ xs\ x)$

new pseudo-constructors

pattern-matching on pseudo-constructors

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$
\begin{aligned}
\mathit{ChF} &:: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set} \\
\mathit{DSet} &:: \alpha\ \mathit{dlist} \Rightarrow \alpha\ \mathit{set} \\
\mathit{RSet} &:: \alpha\ \mathit{srbt} \Rightarrow \alpha\ \mathit{set} \\
\mathit{Compl} &:: \alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set}
\end{aligned}
$$

...

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$\begin{aligned} &ChF &&:: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set \\ &DSet &&:: \alpha\ dlist \Rightarrow \alpha\ set \\ &RSet &&:: \alpha\ srbt \Rightarrow \alpha\ set \\ &Compl &&:: \alpha\ set \Rightarrow \alpha\ set \end{aligned}$$

...

pattern-match on pseudo-constructors

$$\begin{aligned} x \in ChF\ P &= P\ x \\ x \in DSet\ ds &= dmemb\ (op =)\ ds\ x \\ x \in RSet\ rs &= rmemb\ (op <)\ rs\ x \\ x \in Compl\ A &= \neg\ (x \in A) \end{aligned}$$

...

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$\begin{array}{lll} \mathit{ChF} & :: & (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set} \\ \mathit{DSet} & :: & \alpha\ \mathit{dlist} \Rightarrow \alpha\ \mathit{set} \\ \mathit{RSet} & :: & \alpha\ \mathit{srbt} \Rightarrow \alpha\ \mathit{set} \\ \mathit{Compl} & :: & \alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set} \\ \ldots & & \end{array}$$

pattern-match on pseudo-constructors

$$\begin{array}{ll} x \in \mathit{ChF}\ P & = P\ x \\ x \in \mathit{DSet}\ \mathit{ds} & = \mathit{dmemb}\ (\mathit{op} =)\ \mathit{ds}\ x \\ x \in \mathit{RSet}\ \mathit{rs} & = \mathit{rmemb}\ (\mathit{op} <)\ \mathit{rs}\ x \\ x \in \mathit{Compl}\ A & = \neg\ (x \in A) \\ \ldots & \end{array}$$

How do we compare elements?
What if there is no order?

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$\begin{aligned}
ChF &:: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set\\
DSet &:: \alpha\ dlist \Rightarrow \alpha\ set\\
RSet &:: \alpha\ srbt \Rightarrow \alpha\ set\\
Compl &:: \alpha\ set \Rightarrow \alpha\ set\\
&\ldots
\end{aligned}$$

pattern-match on pseudo-constructors

$$\begin{aligned}
x \in ChF\ P &= P\ x\\
x \in DSet\ ds &= dmemb\ (op =)\ ds\ x\\
x \in RSet\ rs &= rmemb\ (op <)\ rs\ x\\
x \in Compl\ A &= \neg\ (x \in A)\\
&\ldots
\end{aligned}$$

How do we compare elements?
What if there is no order?

2. type classes with **optional** operations

class *corder* $=$

fixes $corder :: ((\alpha \Rightarrow \alpha \Rightarrow bool) \times (\alpha \Rightarrow \alpha \Rightarrow bool))$ **option**

assumes $corder = Some\ (le, lt) \Longrightarrow class.linorder\ le\ lt$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$\begin{aligned}
ChF &:: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set \\
DSet &:: \alpha\ dlist \Rightarrow \alpha\ set \\
RSet &:: \alpha\ srbt \Rightarrow \alpha\ set \\
Compl &:: \alpha\ set \Rightarrow \alpha\ set \\
&\ldots
\end{aligned}$$

pattern-match on pseudo-constructors

$$\begin{aligned}
x \in ChF\ P &= P\ x \\
x \in DSet\ ds &= dmemb\ (op =)\ ds\ x \\
x \in RSet\ rs &= rmemb\ (op <)\ rs\ x \\
x \in Compl\ A &= \neg\ (x \in A) \\
&\ldots
\end{aligned}$$

How do we compare elements?
What if there is no order?

2. type classes with **optional** operations

class *corder* =

fixes $corder :: ((\alpha \Rightarrow \alpha \Rightarrow bool) \times (\alpha \Rightarrow \alpha \Rightarrow bool))$ **option**

assumes $corder = Some\ (le, lt) \Longrightarrow class.linorder\ le\ lt$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

   $ChF :: (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$
   $DSet :: \alpha\ dlist \Rightarrow \alpha\ set$
   $RSet :: \alpha\ srbt \Rightarrow \alpha\ set$
   $Compl :: \alpha\ set \Rightarrow \alpha\ set$
   ...

   pattern-match on pseudo-constructors

   $x \in ChF\ P = P\ x$
   $x \in DSet\ ds = dmemb\ (op =)\ ds\ x$
   $x \in RSet\ rs = rmemb\ (op <)\ rs\ x$
   $x \in Compl\ A = \neg\ (x \in A)$
   ...

   How do we compare elements?
   What if there is no order?

2. type classes with **optional** operations

   class *corder* =
   fixes $corder :: ((\alpha \Rightarrow \alpha \Rightarrow bool) \times (\alpha \Rightarrow \alpha \Rightarrow bool))$ **option**
   assumes $corder = Some\ (le, lt) \Longrightarrow class.linorder\ le\ lt$

$insert\ x\ (RSet\ rbt) =$ **case corder of**
$Some\ (\_, \mathrm{lt}) \Rightarrow RSet\ (rinsert\ \mathrm{lt}\ x\ rbt)$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

   $ChF \;::\; (\alpha \Rightarrow bool) \Rightarrow \alpha\ set$
   $DSet \;::\; \alpha\ dlist \Rightarrow \alpha\ set$
   $RSet \;::\; \alpha\ srbt \Rightarrow \alpha\ set$
   $Compl \;::\; \alpha\ set \Rightarrow \alpha\ set$
   ...

   pattern-match on pseudo-constructors

   $x \in ChF\ P = P\ x$
   $x \in DSet\ ds = dmemb\ (op =)\ ds\ x$
   $x \in RSet\ rs = rmemb\ (op <)\ rs\ x$
   $x \in Compl\ A = \neg\ (x \in A)$
   ...

   How do we compare elements?
   What if there is no order?

2. type classes with **optional** operations

   class *corder* =
   fixes $corder :: ((\alpha \Rightarrow \alpha \Rightarrow bool) \times (\alpha \Rightarrow \alpha \Rightarrow bool))$ **option**
   assumes $corder = Some\ (le, lt) \Longrightarrow class.linorder\ le\ lt$

$insert\ x\ (RSet\ rbt) =$ **case** *corder* **of**
$Some\ (\_, lt) \Rightarrow RSet\ (rinsert\ lt\ x\ rbt)$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation

$$
\begin{aligned}
\mathit{ChF} &:: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set} \\
\mathit{DSet} &:: \alpha\ \mathit{dlist} \Rightarrow \alpha\ \mathit{set} \\
\mathit{RSet} &:: \alpha\ \mathit{srbt} \Rightarrow \alpha\ \mathit{set} \\
\mathit{Compl} &:: \alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set} \\
&\ldots
\end{aligned}
$$

pattern-match on pseudo-constructors

$$
\begin{aligned}
x \in \mathit{ChF}\ P &= P\ x \\
x \in \mathit{DSet}\ \mathit{ds} &= \mathit{dmemb}\ (\mathit{op} =)\ \mathit{ds}\ x \\
x \in \mathit{RSet}\ \mathit{rs} &= \mathit{rmemb}\ (\mathit{op} <)\ \mathit{rs}\ x \\
x \in \mathit{Compl}\ A &= \neg\ (x \in A) \\
&\ldots
\end{aligned}
$$

How do we compare elements?
What if there is no order?

2. type classes with **optional** operations

class *corder* =

fixes $\mathit{corder} :: ((\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}) \times (\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}))$ **option**

assumes $\mathit{corder} = \mathit{Some}\ (\mathit{le}, \mathit{lt}) \Longrightarrow \mathit{class.linorder}\ \mathit{le}\ \mathit{lt}$

requires run-time test:

$\mathit{insert}\ x\ (\mathit{RSet}\ \mathit{rbt}) =$ **case corder of None** $\Rightarrow$ **error** ...

$\mid \mathit{Some}\ (\_, \mathrm{lt}) \Rightarrow \mathit{RSet}\ (\mathit{rinsert}\ \mathrm{lt}\ x\ \mathit{rbt})$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\begin{aligned} \{\} = \ & \mathit{case\ corder\ of\ Some\ \_} \Rightarrow \mathit{RSet\ rempty} \\ & \mid \mathit{None} \Rightarrow \mathit{case\ ceq\ of\ Some\ \_} \Rightarrow \mathit{DSet\ dempty} \\ & \mid \mathit{None} \Rightarrow \mathit{ChF}\ (\lambda\_.\ \mathit{False}) \end{aligned}$$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\{\} = \textit{case corder of Some } \_ \Rightarrow \textit{RSet rempty}$$
$$\mid \textit{None} \Rightarrow \textit{case ceq of Some } \_ \Rightarrow \textit{DSet dempty}$$
$$\mid \textit{None} \Rightarrow \textit{ChF } (\lambda\_.\ \textit{False})$$

$\Rightarrow$ run-time tests always succeed

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\begin{aligned}\{\} = {} & \mathit{case\ corder\ of\ Some\ \_} \Rightarrow \mathit{RSet\ rempty} \\ & \mid \mathit{None} \Rightarrow \mathit{case\ ceq\ of\ Some\ \_} \Rightarrow \mathit{DSet\ dempty} \\ & \mid \mathit{None} \Rightarrow \mathit{ChF}\ (\lambda \_.\ \mathit{False})\end{aligned}$$

4. handle binary operators

$$\mathcal{O}(\mathbf{n}^2)\ \text{cases} \begin{cases} \mathit{RSet}\ rs_1 \cap \mathit{RSet}\ rs_2 = \dots \\ \mathit{RSet}\ rs_1 \cap \mathit{DSet}\ ds_2 = \dots \\ \mathit{RSet}\ rs_1 \cap \mathit{ChF}\ P_2 = \dots \\ \mathit{RSet}\ rs_1 \cap \mathit{Compl}\ A = \dots \\ \mathit{DSet}\ ds_1 \cap \mathit{RSet}\ rs_2 = \dots \\ \mathit{DSet}\ ds_1 \cap \mathit{DSet}\ ds_2 = \dots \\ \qquad \vdots \qquad\qquad \vdots \end{cases}$$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\begin{aligned}\{\} = & \textit{case corder of Some } \_ \Rightarrow \textit{RSet rempty} \\ & \mid \textit{None} \Rightarrow \textit{case ceq of Some } \_ \Rightarrow \textit{DSet dempty} \\ & \mid \textit{None} \Rightarrow \textit{ChF } (\lambda\_. \textit{ False})\end{aligned}$$

4. handle binary operators

$\mathcal{O}(\mathbf{n})$ cases

$$\begin{cases} \textit{RSet } rs_1 \cap A = \textit{RSet } (\textit{rfilter } (\lambda x.\ x \in A)\ rs_1) \\ \textit{DSet } ds_1 \cap A = \textit{DSet } (\textit{dfilter } (\lambda x.\ x \in A)\ ds_1) \\ \quad \vdots \qquad\qquad \vdots \end{cases}$$

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\begin{aligned} \{\} = {} & \mathit{case\ corder\ of\ Some}\ \_ \Rightarrow \mathit{RSet\ rempty} \\ & \mid \mathit{None} \Rightarrow \mathit{case\ ceq\ of\ Some}\ \_ \Rightarrow \mathit{DSet\ dempty} \\ & \mid \mathit{None} \Rightarrow \mathit{ChF}\ (\lambda\_.\ \mathit{False}) \end{aligned}$$

4. handle binary operators with sequential pattern matching

$$\mathcal{O}(\mathbf{n})\ \text{cases} \begin{cases} \mathit{RSet}\ rs_1 \cap A = \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs_1) \\ A \cap \mathit{RSet}\ rs_2 = \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs_2) \\ \mathit{DSet}\ ds_1 \cap A = \mathit{DSet}\ (\mathit{dfilter}\ (\lambda x.\ x \in A)\ ds_1) \\ \quad \vdots \qquad \vdots \end{cases}$$

precedes

Multiple implementations for a container

1. 1 pseudo-constructor for each implementation
2. type classes with optional operations
3. choose a suitable implementation based on available operations

$$\begin{aligned}\{\} = {} & \mathit{case\ corder\ of\ Some}\ \_ \Rightarrow \mathit{RSet\ rempty} \\ & \mid \mathit{None} \Rightarrow \mathit{case\ ceq\ of\ Some}\ \_ \Rightarrow \mathit{DSet\ dempty} \\ & \mid \mathit{None} \Rightarrow \mathit{ChF}\ (\lambda\_.\ \mathit{False})\end{aligned}$$

4. handle binary operators with sequential pattern matching

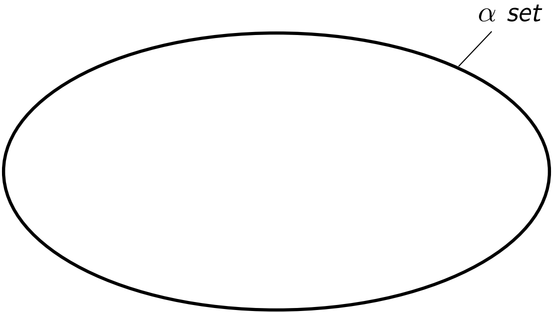
$\mathcal{O}(\mathbf{n})$ cases + $\mathbf{m}$

$$\begin{cases} \mathit{RSet}\ rs_1 \cap \mathit{RSet}\ rs_2 = \mathit{RSet}\ (\mathit{rint}\ rs_1\ rs_2) \\ \mathit{RSet}\ rs_1 \cap A = \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs_1) \\ A \cap \mathit{RSet}\ rs_2 = \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs_2) \\ \mathit{DSet}\ ds_1 \cap A = \mathit{DSet}\ (\mathit{dfilter}\ (\lambda x.\ x \in A)\ ds_1) \\ \quad\vdots \qquad\qquad \vdots \end{cases}$$

precedes

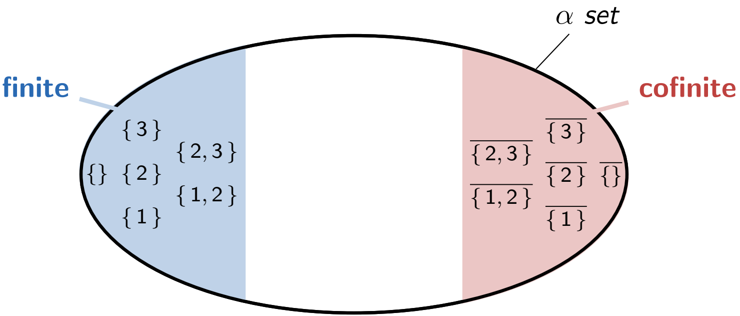
Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



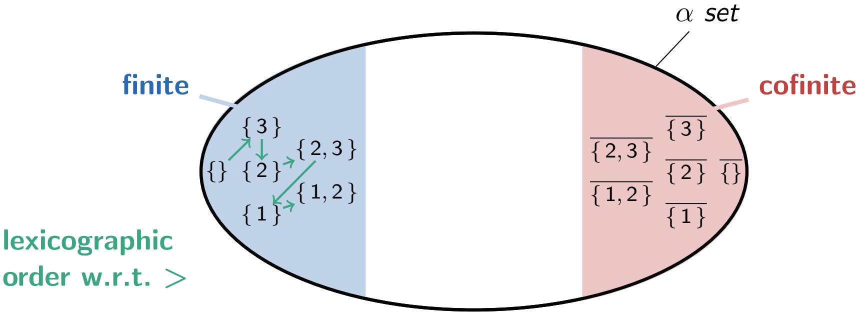
Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



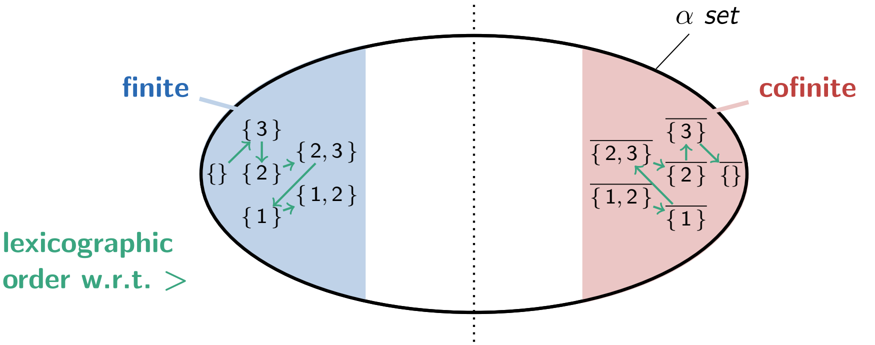
Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



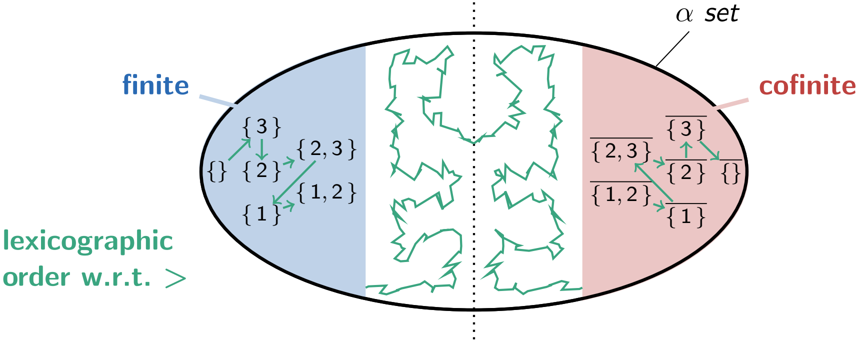
Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



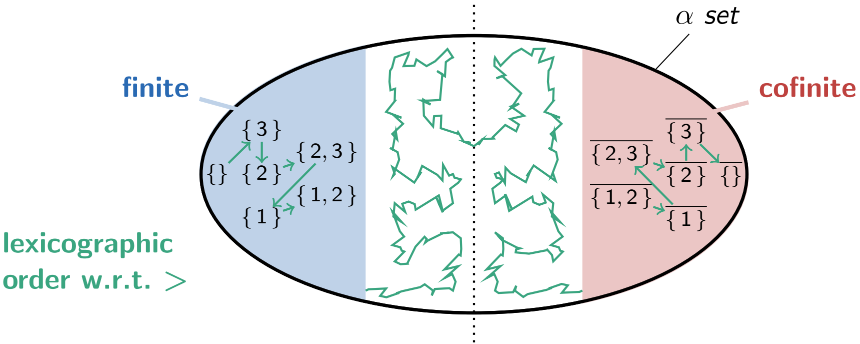
Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



$\{\} \sqsubseteq A$ and $A \sqsubseteq \overline{\{\}}$

$\overline{A} \sqsubseteq \overline{B}$ iff $B \sqsubseteq A$

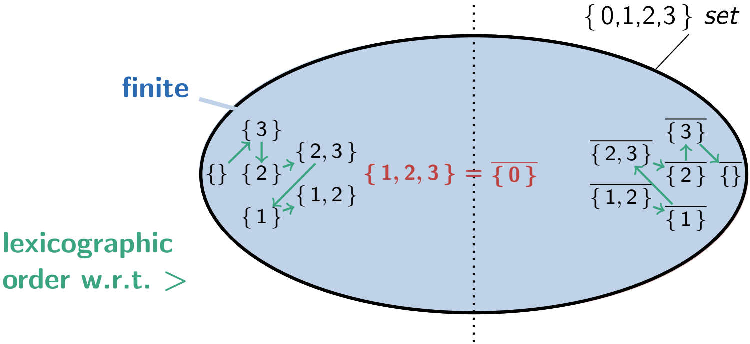
If $F \subseteq F'$, then $F \sqsubseteq F'$

If α is infinite, then $F \sqsubseteq \overline{F'}$

(both) for F, F' finite

Nesting containers

α *set set* as a search tree requires computable linear order $\sqsubseteq$ on α *set*



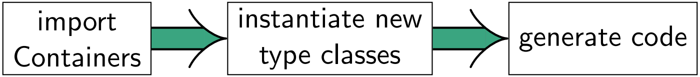
$\{\} \sqsubseteq A$ and $A \sqsubseteq \overline{\{\}}$

$\overline{A} \sqsubseteq \overline{B}$ iff $B \sqsubseteq A$

If $F \subseteq F'$, then $F \sqsubseteq F'$
If α is infinite, then $F \sqsubseteq \overline{F'}$
} for F, F' finite

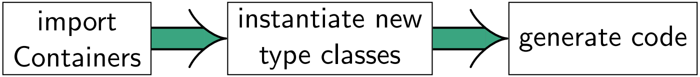
Evaluation

Easy to use: case study with Java interpreter



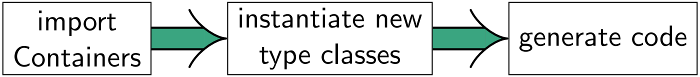
Evaluation

Easy to use: case study with Java interpreter ✓



Evaluation

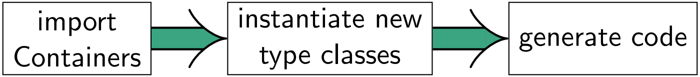
Easy to use: case study with Java interpreter ✓



Extensible, flexible, nestable: ✓

Evaluation

Easy to use: case study with Java interpreter ✓



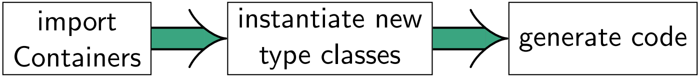
Extensible, flexible, nestable: ✓

Efficient

1. ICF benchmark: as fast as the ICF and pure RBTs (*int set*)
2. nested sets: confirm complexity of $\sqsubset$: $\mathcal{O}(n^d)$
3. Java interpreter:
   type classes cause overhead for large types (tune equations)

Evaluation

Easy to use: case study with Java interpreter ✓



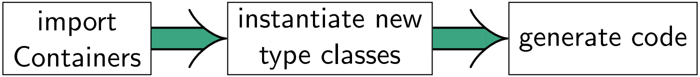
Extensible, flexible, nestable: ✓

Efficient ✓

1. ICF benchmark: as fast as the ICF and pure RBTs (*int set*)
2. nested sets: confirm complexity of $\sqsubset$: $\mathcal{O}(n^d)$
3. Java interpreter:
   type classes cause overhead for large types (tune equations)

Evaluation

Easy to use: case study with Java interpreter ✓



Extensible, flexible, nestable: ✓

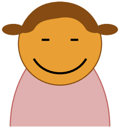
Efficient ✓

1. ICF benchmark: as fast as the ICF and pure RBTs (*int set*)
2. nested sets: confirm complexity of $\sqsubset$: $\mathcal{O}(n^d)$
3. Java interpreter:
   type classes cause overhead for large types (tune equations)

Limitation: folding over a containers requires commutative operator
refinement in the code generator is logically irrelevant

Summary

Light-weight containers



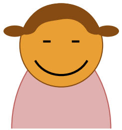
- approach to efficiently implement containers
- light-weight: refinement during code generation
- extensible, flexible, nestable

- Available in the Archive of Formal Proofs

  `http://afp.sourceforge.net/entries/Containers.shtml`

Summary

Light-weight containers



- approach to efficiently implement containers
- light-weight: refinement during code generation
- extensible, flexible, nestable

- Available in the Archive of Formal Proofs

  `http://afp.sourceforge.net/entries/Containers.shtml`

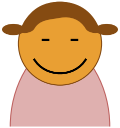
Essential code generator features

- incremental declarations
- data refinement with invariants
- type classes
- sequential pattern matching

Summary

Light-weight containers

- approach to efficiently implement containers
- light-weight: refinement during code generation
- extensible, flexible, nestable



- Available in the Archive of Formal Proofs

  `http://afp.sourceforge.net/entries/Containers.shtml`

Essential code generator features

- incremental declarations
- data refinement with invariants
- type classes
- sequential pattern matching

Future work

- cover more containers and implementations
- combine with Autoref/ICF