Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions

Ioannis T. Kassios

ykass@cs.toronto.edu

Department of Computer Science

University of Toronto

Framing and specification attributes

- *Framing specifications* list all the attributes that are modified by an operation:

  modifies x, y, z

Framing and specification attributes

- *Framing specifications* list all the attributes that are modified by an operation:

  modifies x, y, z

- *Specification attributes* are used for information hiding:

  public spec attr L

  private prog attr *head*

  $L = \lambda i \cdot \ \mathit{head}\mathbf{.}[\mathit{next}]^i\mathbf{.}\mathit{val}$

Framing and specification attributes

- *Framing specifications* list all the attributes that are modified by an operation:

  modifies x, y, z

- *Specification attributes* are used for information hiding:

  public spec attr L

  private prog attr *head*

  $L = \lambda i \cdot \; \mathit{head}.[\mathit{next}]^i.\mathit{val}$

- Framing on specification attributes

  modifies L

  means

  modifies *head* , *head.val* , *head.next* , *head.next.val* , ...

  license to modify $L \Rightarrow$ license to modify all attributes on which L *is known to depend*

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y
- *Abstract aliasing*: the representation of y might accidentally share heap locations with the representation of L
  - abstract reasoning may be unsound

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y
- *Abstract aliasing*: the representation of y might accidentally share heap locations with the representation of L
  - abstract reasoning may be unsound
- Solutions:
  - no support for pointers

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y
- *Abstract aliasing*: the representation of y might accidentally share heap locations with the representation of L
  - abstract reasoning may be unsound
- Solutions:
  - no support for pointers or encapsulation

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y
- *Abstract aliasing*: the representation of y might accidentally share heap locations with the representation of L
  - abstract reasoning may be unsound
- Solutions:
  - no support for pointers or encapsulation or framing

Abstract aliasing

- Consider operation P with specification

  modifies L

  and a client with a specification attribute y
  - abstract reasoning predicts that P preserves y
- *Abstract aliasing*: the representation of y might accidentally share heap locations with the representation of L
  - abstract reasoning may be unsound
- Solutions:
  - no support for pointers or encapsulation or framing
  - forbid abstract aliasing: (Leino, Nelson 2002), Universes (Müller 2002), Boogie (Leino, Müller 2004)

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)
  - inflexibility (useful designs excluded)

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)
  - inflexibility (useful designs excluded)
- Proposed solution: *Dynamic Frames*

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)
  - inflexibility (useful designs excluded)
- Proposed solution: *Dynamic Frames*
  - idea: make (absence of) abstract aliasing *expressible*

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)
  - inflexibility (useful designs excluded)
- Proposed solution: *Dynamic Frames*
  - idea: make (absence of) abstract aliasing *expressible*
  - no new formalism (d.f. = special case of spec. attributes)

The Idea behind Dynamic Frames

- Forbidding abstract aliasing has drawbacks
  - complication (new formalisms introduced)
  - inflexibility (useful designs excluded)
- Proposed solution: *Dynamic Frames*
  - idea: make (absence of) abstract aliasing *expressible*
  - no new formalism (d.f. = special case of spec. attributes)
  - no programming restrictions

Open Expressions

- *Open expression* = a named expression with possibly free variables
  - used as a syntactic convenience

Open Expressions

- *Open expression* = a named expression with possibly free variables
  - used as a syntactic convenience
- Example:

$$
\begin{aligned}
E &= 2x \\
\left(\sum x \in \{1,2\} \cdot\ E\right) &= 2 \cdot 1 + 2 \cdot 2 = 6
\end{aligned}
$$

Open Expressions

- *Open expression* = a named expression with possibly free variables
  - used as a syntactic convenience
- Example:

$$
\begin{aligned}
E &= 2x \\
\left(\sum x \in \{1, 2\} \cdot\ E\right) &= 2 \cdot 1 + 2 \cdot 2 = 6
\end{aligned}
$$

- Notation $E(t/x)$ stands for substitution:

$$
E(4/x) = 2 \cdot 4 = 8
$$

Notation

- *Domain restriction*: $(f \triangleright D) \; = \; \lambda x \in \text{Dom } f \cap D \cdot \; f \; x$

Notation

- *Domain restriction*: $(f \triangleright D) \; = \; \lambda x \in \textsf{Dom}\ f \cap D \cdot \; f\ x$
- *Range*: $\{l, ..u\} \; = \; \{x \in \mathbb{Z} \cdot \; l \leq x < u\}$

Notation

- *Domain restriction*: $(f \triangleright D) = \lambda x \in \text{Dom } f \cap D \cdot f\ x$
- *Range*: $\{l, ..u\} = \{x \in \mathbb{Z} \cdot l \leq x < u\}$
- *Lists*: functions with domain $\{0, ..n\}$
  - construction: $[..; ..; ....]$
  - set of lists: X^*
  - size: $\#L$
  - concatenation: $L^\frown M$

Notation

- *Domain restriction*: $(f \rhd D) \;=\; \lambda x \in \text{Dom}\ f \cap D \cdot\ f\ x$
- *Range*: $\{l, ..u\} \;=\; \{x \in \mathbb{Z} \cdot\ l \leq x < u\}$
- *Lists*: functions with domain $\{0, ..n\}$
  - construction: $[..; ..; ....]$
  - set of lists: X^*
  - size: $\#L$
  - concatenation: $L \frown M$
- List of disjoint sets:

$$(\mathit{disjoint}\ L) \;=\; (\forall i, j \cdot\ i \neq j \;\Rightarrow\; L\ i \cap L j = \emptyset)$$

Basics - I

- *Set of locations*: Loc
  - *region* = subset of Loc
- *Set of states*: Σ (mappings from locations to values)

Basics - I

- *Set of locations*: Loc
  - *region* = subset of Loc
- *Set of states*: Σ (mappings from locations to values)
- Final value of E:

$$E' \;=\; E(\sigma'/\sigma)$$

Basics - I

- *Set of locations*: Loc
  - *region* = subset of Loc
- *Set of states*: Σ (mappings from locations to values)
- Final value of E:

$$E' \;=\; E(\sigma'/\sigma)$$

- *Specification variable* = open expression on σ

Basics - I

- *Set of locations*: Loc
  - *region* = subset of Loc
- *Set of states*: Σ (mappings from locations to values)
- Final value of E:

$$E' \ = \ E(\sigma'/\sigma)$$

- *Specification variable* = open expression on σ
  - example: $Unused = Loc - \mathrm{Dom}\ \sigma$

Basics - I

- *Set of locations*: Loc
  - *region* = subset of Loc
- *Set of states*: Σ (mappings from locations to values)
- Final value of E:

$$E' \;=\; E(\sigma'/\sigma)$$

- *Specification variable* = open expression on σ
  - example: $Unused = Loc - \mathrm{Dom}\ \sigma$
- *Program variable* x:

$$x \;=\; \sigma(addr\_x)$$

  - $addr\_x$ is the *address of* x

Basics - II

- *Imperative specification* = boolean expression on state-valued σ, σ'
  - *program* = special case of imperative specification

Basics - II

- *Imperative specification* = boolean expression on state-valued σ, σ'
  - *program* = special case of imperative specification
- *Module* = collection of declarations and axioms
  - **module** introduces the module
  - **spec var** introduces a specification variable
  - **prog var** introduces a program variable
  - **proc** introduces a procedure specification
  - **import** imports names / axioms from another module

Basics - II

- *Imperative specification* = boolean expression on state-valued σ, σ'
  - *program* = special case of imperative specification
- *Module* = collection of declarations and axioms
  - **module** introduces the module
  - **spec var** introduces a specification variable
  - **prog var** introduces a program variable
  - **proc** introduces a procedure specification
  - **import** imports names / axioms from another module
- M refines N iff
  - names of $M \subseteq$ names of N
  - axiom of $M \Rightarrow$ axiom of N

Basics: Example

module *RationalSpec*

spec var $\mathit{rat\_inv} \in \mathsf{Bool}\ ,\ \mathit{rat}$

$\mathit{rat\_inv} \Rightarrow \mathit{rat} \in \mathbb{Q}$

proc $\mathit{double}() \cdot\ \mathit{rat\_inv}\ \Rightarrow\ \mathit{rat}' = 2 \times \mathit{rat}\ \wedge\ \mathit{rat\_inv}'$

end module

Basics: Example

module *RationalSpec*

spec var $\mathit{rat\_inv} \in \mathsf{Bool}\ ,\ \mathit{rat}$

$\mathit{rat\_inv} \Rightarrow \mathit{rat} \in \mathbb{Q}$

proc $\mathit{double}() \cdot\ \mathit{rat\_inv}\ \Rightarrow\ \mathit{rat}' = 2 \times \mathit{rat}\ \wedge\ \mathit{rat\_inv}'$

end module

module *RationalImpl*

prog var $\mathit{nom}, \mathit{denom}$

spec var $\mathit{rat\_inv}\ =\ (\mathit{nom} \in \mathbb{Z}\ \wedge\ \mathit{denom} \in \mathbb{N} - \{0\})$

spec var $\mathit{rat} = \mathit{nom}/\mathit{denom}$

proc $\mathit{double}() \cdot\ \mathit{nom} := 2 \times \mathit{nom}$

end module

Dynamic Frames

- Framing on regions: if R is a region:
  - preservation: $\Xi R \;=\; (\sigma' \triangleright R \;=\; \sigma \triangleright R)$
  - modification: $\Delta R \;=\; \Xi(\text{Dom}\ \sigma - R)$

Dynamic Frames

- Framing on regions: if R is a region:
  - preservation: $\Xi R \;=\; (\sigma' \triangleright R \;=\; \sigma \triangleright R)$
  - modification: $\Delta R \;=\; \Xi(\text{Dom}\ \sigma - R)$
- *Dynamic frame* f = specification variable whose value is a set of allocated locations:

$$f \subseteq \text{Dom}\ \sigma$$

Dynamic Frames

- Framing on regions: if R is a region:
  - preservation: $\Xi R \;=\; (\sigma' \triangleright R \;=\; \sigma \triangleright R)$
  - modification: $\Delta R \;=\; \Xi(\text{Dom}\,\sigma - R)$
- *Dynamic frame* f = specification variable whose value is a set of allocated locations:

$$f \subseteq \text{Dom}\,\sigma$$

- Variable framing: if f is a dynamic frame and v is a spec. variable:

$$(f \;\mathbf{frames}\; v) \;=\; (\forall \sigma' \in \Sigma \cdot \; \Xi f \Rightarrow v' = v)$$

Dynamic Frames: Example

module *RationalSpec*

spec var $rat\_inv \in \mathsf{Bool}\ ,\ rat$

$rat\_inv \ \Rightarrow\ rat \in \mathbb{Q}$

proc $double() \cdot\ rat\_inv \ \Rightarrow\ rat' = 2 \times rat \ \wedge\ rat\_inv'$

end module

Dynamic Frames: Example

module *RationalSpec*

spec var $\mathit{rat\_inv} \in \mathsf{Bool}$, rat , $\mathit{rat\_rep}$

$\mathit{rat\_inv} \;\Rightarrow\; \mathit{rat} \in \mathbb{Q}$

proc $\mathit{double}() \cdot\; \mathit{rat\_inv} \;\Rightarrow\; \mathit{rat}' = 2 \times \mathit{rat} \;\wedge\; \mathit{rat\_inv}'$

end module

Dynamic Frames: Example

module *RationalSpec*

spec var $rat\_inv \in \mathsf{Bool}$, rat , rat_rep

$rat\_inv \Rightarrow rat \in \mathbb{Q} \wedge rat\_rep \subseteq \mathsf{Dom}\ \sigma \wedge rat\_rep$ **frames** (rat, rat_inv)

proc $double() \cdot rat\_inv \Rightarrow rat' = 2 \times rat \wedge rat\_inv'$

end module

Dynamic Frames: Example

module *RationalSpec*

spec var $rat\_inv \in \mathsf{Bool}$, rat , rat_rep

$rat\_inv \;\Rightarrow\; rat \in \mathbb{Q} \wedge rat\_rep \subseteq \mathsf{Dom}\,\sigma \;\wedge\; rat\_rep$ **frames** (rat, rat_inv)

proc $double() \cdot \; rat\_inv \;\Rightarrow\; rat' = 2 \times rat \;\wedge\; rat\_inv' \wedge \Delta rat\_rep$

end module

Dynamic Frames: Example

module *RationalSpec*

spec var $rat\_inv \in \mathsf{Bool}$, rat , rat_rep

$rat\_inv \;\Rightarrow\; rat \in \mathbb{Q} \wedge rat\_rep \subseteq \mathsf{Dom}\ \sigma \;\wedge\; rat\_rep$ **frames** (rat, rat_inv)

proc $double() \cdot\; rat\_inv \;\Rightarrow\; rat' = 2 \times rat \;\wedge\; rat\_inv' \wedge \Delta rat\_rep$

end module

module *RationalImpl*

prog var $nom, denom$

spec var $rat\_inv \;=\; (nom \in \mathbb{Z} \;\wedge\; denom \in \mathbb{N} - \{0\})$

spec var $rat = nom/denom$

proc $double() \cdot\; nom := 2 \times nom$

end module

Dynamic Frames: Example

module *RationalSpec*

  spec var $rat\_inv \in \mathsf{Bool}$, rat , rat_rep

  $rat\_inv \;\Rightarrow\; rat \in \mathbb{Q} \wedge rat\_rep \subseteq \mathsf{Dom}\;\sigma \;\wedge\; rat\_rep$ **frames** (rat, rat_inv)

  proc $double() \cdot\; rat\_inv \;\Rightarrow\; rat' = 2 \times rat \;\wedge\; rat\_inv' \wedge \Delta rat\_rep$

end module

module *RationalImpl*

  prog var $nom, denom$

  spec var $rat\_inv \;=\; (nom \in \mathbb{Z} \;\wedge\; denom \in \mathbb{N} - \{0\})$

  spec var $rat = nom/denom$

  spec var $rat\_rep = \{addr\_nom, addr\_denom\}$

  proc $double() \cdot\; nom := 2 \times nom$

end module

Dynamic Frames: Example

module *RationalSpec*

spec var $rat\_inv \in \mathsf{Bool}$, rat , rat_rep

$rat\_inv \;\Rightarrow\; rat \in \mathbb{Q} \wedge rat\_rep \subseteq \mathsf{Dom}\,\sigma \;\wedge\; rat\_rep$ **frames** (rat, rat_inv)

proc $double() \cdot\; rat\_inv \;\Rightarrow\; rat' = 2 \times rat \;\wedge\; rat\_inv' \wedge \Delta rat\_rep$

end module

module *RationalImpl*

prog var $nom, denom$

spec var $rat\_inv \;=\; (nom \in \mathbb{Z} \;\wedge\; denom \in \mathbb{N} - \{0\})$

spec var $rat = nom/denom$

spec var $rat\_rep = \{addr\_nom, addr\_denom\}$

proc $double() \cdot\; nom := 2 \times nom$

end module

in general: dynamic frames vary with state

Independence

- *Independence* (absence of abstract aliasing) = disjointness of frames

Independence

- *Independence* (absence of abstract aliasing) = disjointness of frames
- Modification of frame f guarantees preservation of any independent specification variable y:

$$g \textbf{ frames } y \ \wedge \ \textit{disjoint}[f;g] \ \wedge \ \Delta f \quad \Rightarrow \quad y' = y$$

Independence

- *Independence* (absence of abstract aliasing) = disjointness of frames
- Modification of frame f guarantees preservation of any independent specification variable y:

$$g\ \mathbf{frames}\ y\ \wedge\ \mathit{disjoint}[f;g]\ \wedge\ \Delta f\ \ \Rightarrow\ \ y' = y$$

  - modularity: the implementer of Δf does not need to know g, y

Independence: Example

A client of *RationalSpec*:

module *ZSpec*

import *RationalSpec*

spec var $z\_inv \in \mathsf{Bool}$, z, z_rep

$z\_inv \;\Rightarrow\; rat\_inv \wedge z\_rep \subseteq \mathsf{Dom}\,\sigma$

$z\_inv \;\Rightarrow\; z\_rep$ **frames** $z \wedge disjoint[z\_rep; rat\_rep]$

...

end module

Independence: Example

A client of *RationalSpec*:

$$
\begin{array}{l}
\textbf{module } \mathit{ZSpec} \\
\quad \textbf{import } \mathit{RationalSpec} \\
\quad \textbf{spec var } z\_inv \in \mathsf{Bool}\ ,\ z,\ z\_rep \\
\quad z\_inv \;\Rightarrow\; rat\_inv \wedge z\_rep \subseteq \mathsf{Dom}\,\sigma \\
\quad z\_inv \;\Rightarrow\; z\_rep \textbf{ frames } z \wedge disjoint[z\_rep; rat\_rep] \\
\quad \ldots \\
\textbf{end module}
\end{array}
$$

we can prove:

$$double() \wedge z\_inv \;\Rightarrow\; z' = z$$

because: $double() \;\Rightarrow\; \Delta rat\_rep$

Self-framing

- Disjointness of dynamic frames must be preserved

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f
- *Swinging pivots* specification (Leino, Nelson 2002):

$$f' \subseteq f \cup \mathit{Unused}$$

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f
- *Swinging pivots* specification (Leino, Nelson 2002):

$$f' \subseteq f \cup \mathit{Unused}$$

- Swinging pivots on f ensures preservation of disjointness from any unknown self-framing frame g:

$$\Delta f \wedge f' \subseteq f \cup \mathit{Unused}$$

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f
- *Swinging pivots* specification (Leino, Nelson 2002):

$$f' \subseteq f \cup \mathit{Unused}$$

- Swinging pivots on f ensures preservation of disjointness from any unknown self-framing frame g:

$$\Delta f \wedge f' \subseteq f \cup \mathit{Unused} \wedge \mathit{disjoint}[f;g]$$

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f
- *Swinging pivots* specification (Leino, Nelson 2002):

$$f' \subseteq f \cup \mathit{Unused}$$

- Swinging pivots on f ensures preservation of disjointness from any unknown self-framing frame g:

$$\Delta f \wedge f' \subseteq f \cup \mathit{Unused} \wedge \mathit{disjoint}[f; g] \;\Rightarrow\; (\mathit{disjoint}[f; g])'$$

Self-framing

- Disjointness of dynamic frames must be preserved
- Dynamic frames are self-framed: f **frames** f
- *Swinging pivots* specification (Leino, Nelson 2002):

$$f' \subseteq f \cup \mathit{Unused}$$

- Swinging pivots on f ensures preservation of disjointness from any unknown self-framing frame g:

$$\Delta f \wedge f' \subseteq f \cup \mathit{Unused} \wedge \mathit{disjoint}[f; g] \;\Rightarrow\; (\mathit{disjoint}[f; g])'$$

- More generally:

$$\Delta f \wedge f' \subseteq f \cup \mathit{Unused} \cup h \wedge \mathit{disjoint}[f \cup h \,;\, g] \;\Rightarrow\; (\mathit{disjoint}[f; g])'$$

Objects

- Set of object references: $\mathcal{O}$

Objects

- Set of object references: $\mathcal{O}$
- *Specification attribute* = open expression on $\sigma \in \Sigma$ and $\mathit{self} \in \mathcal{O}$
  - *program attribute*: special case

Objects

- Set of object references: $\mathcal{O}$
- *Specification attribute* = open expression on $\sigma \in \Sigma$ and $\mathit{self} \in \mathcal{O}$
  - *program attribute*: special case
- Dot notation:

$$(p{\bullet}E) \;=\; E(p/\mathit{self})$$

Objects

- Set of object references: $\mathcal{O}$
- *Specification attribute* = open expression on $\sigma \in \Sigma$ and $\mathit{self} \in \mathcal{O}$
  - *program attribute*: special case
- Dot notation:

$$(p\bullet E) \;=\; E(p/\mathit{self})$$

- Chains:

$$[E]^0 = \mathit{self}$$
$$[E]^{n+1} = [E]^n\bullet E$$

Objects

- Set of object references: $\mathcal{O}$
- *Specification attribute* = open expression on $\sigma \in \Sigma$ and $\mathit{self} \in \mathcal{O}$
  - *program attribute*: special case
- Dot notation:

$$(p\bullet E) \;=\; E(p/\mathit{self})$$

- Chains:

$$[E]^0 = \mathit{self}$$
$$[E]^{n+1} = [E]^n\bullet E$$

- Null reference: *null*

Classes

- Class = subset of $\mathcal{O}$

Classes

- Class = subset of $\mathcal{O}$
- Class specification
  - **class** introduces the class
  - **spec attr** introduces a specification attribute
  - **prog attr** introduces a program attribute
  - **method** introduces a method specification

Classes

- Class = subset of $\mathcal{O}$
- Class specification
  - **class** introduces the class
  - **spec attr** introduces a specification attribute
  - **prog attr** introduces a program attribute
  - **method** introduces a method specification
- In the specification of class C, the identifier *self* is implicitly universally quantified over C

Example: List Specification

module *ListSpec*

class *List*

Example: List Specification

module *ListSpec*

class *List*

spec attr L , *inv* $\in$ Bool , *rep*

inv $\Rightarrow$ $L \in \mathbb{Z}^* \ \wedge\ rep \subseteq \mathsf{Dom}\ \sigma \ \wedge\ rep$ **frames** (L, inv, rep)

Example: List Specification

module *ListSpec*

class *List*

spec attr L , *inv* $\in$ Bool , *rep*

$$inv \;\Rightarrow\; L \in \mathbb{Z}^* \;\wedge\; rep \subseteq \mathsf{Dom}\,\sigma \;\wedge\; rep\ \mathbf{frames}\ (L, inv, rep)$$

method *insert*$(x)\cdot$

$$inv \;\wedge\; x \in \mathbb{Z} \;\Rightarrow\; \Delta rep \;\wedge\; L' = [x]^\frown L \;\wedge\; inv' \;\wedge\; rep' \subseteq rep \cup Unused$$

Example: List Specification

module *ListSpec*

class *List*

$$\textbf{spec attr}\ L\ ,\ \mathit{inv} \in \mathsf{Bool}\ ,\ \mathit{rep}$$

$$\mathit{inv} \;\Rightarrow\; L \in \mathbb{Z}^* \;\wedge\; \mathit{rep} \subseteq \mathsf{Dom}\,\sigma \;\wedge\; \mathit{rep}\ \textbf{frames}\ (L, \mathit{inv}, \mathit{rep})$$

$$\textbf{method}\ \mathit{insert}(x)\cdot$$

$$\mathit{inv} \;\wedge\; x \in \mathbb{Z} \;\Rightarrow\; \Delta \mathit{rep} \;\wedge\; L' = [x]^\frown L \;\wedge\; \mathit{inv}' \;\wedge\; \mathit{rep}' \subseteq \mathit{rep} \cup \mathit{Unused}$$

$$\textbf{method}\ \mathit{paste}(p)\cdot$$

$$\mathit{inv} \;\wedge\; p \in \mathit{List} \;\wedge\; p.\mathit{inv} \;\wedge\; \mathit{disjoint}[\mathit{rep}\ ;\ p.\mathit{rep}]$$

$$\Rightarrow\; \Delta(\mathit{rep} \cup p.\mathit{rep}) \;\wedge\; L' = p.L^\frown L \;\wedge\; \mathit{inv}' \;\wedge\; \mathit{rep}' \subseteq \mathit{rep} \cup \mathit{Unused} \cup p.\mathit{rep}$$

Example: List Specification

module *ListSpec*

class *List*

spec attr L , *inv* $\in$ Bool , *rep*

$$inv \;\Rightarrow\; L \in \mathbb{Z}^* \;\wedge\; rep \subseteq \mathsf{Dom}\,\sigma \;\wedge\; rep\ \mathbf{frames}\ (L, inv, rep)$$

method *insert*$(x)\cdot$

$$inv \;\wedge\; x \in \mathbb{Z} \;\Rightarrow\; \Delta rep \;\wedge\; L' = [x]^\frown L \;\wedge\; inv' \;\wedge\; rep' \subseteq rep \cup Unused$$

method *paste*$(p)\cdot$

$$inv \;\wedge\; p \in List \;\wedge\; p.inv \;\wedge\; disjoint[rep\ ;\ p.rep]$$

$$\Rightarrow\; \Delta(rep \cup p.rep) \;\wedge\; L' = p.L^\frown L \;\wedge\; inv' \;\wedge\; rep' \subseteq rep \cup Unused \cup p.rep$$

end class

end module

Example: List Implementation

module *ListImpl*

class *Node*

prog attr $val, next$

spec attr $inv \; = \; (val \in \mathbb{Z})$

spec attr $rep \; = \; \{addr\_next, addr\_val\}$

end class

class *List*

prog attr *head*

Example: List Implementation

module *ListImpl*

class *Node*

prog attr *val*, *next*

spec attr *inv* $= (val \in \mathbb{Z})$

spec attr *rep* $= \{addr\_next, addr\_val\}$

end class

class *List*

prog attr *head*

spec attr $len = min\{i \in \mathbb{N} \cdot \ head.[next]^i = null\}$

Example: List Implementation

module *ListImpl*

class *Node*

prog attr *val*, *next*

spec attr *inv* $= (val \in \mathbb{Z})$

spec attr *rep* $= \{addr\_next, addr\_val\}$

end class

class *List*

prog attr *head*

spec attr $len = min\{i \in \mathbb{N} \cdot \; head.[next]^i = null\}$

spec attr $L = \lambda i \in \{0, ..len\} \cdot head.[next]^i.val$

Example: List Implementation

module *ListImpl*

class *Node*

prog attr *val*, *next*

spec attr *inv* $=$ $(\mathit{val} \in \mathbb{Z})$

spec attr *rep* $=$ $\{\mathit{addr\_next}, \mathit{addr\_val}\}$

end class

class *List*

prog attr *head*

spec attr *len* $= \mathit{min}\{i \in \mathbb{N} \cdot \ \mathit{head}\textbf{.}[\mathit{next}]^i = \mathit{null}\}$

spec attr $L = \lambda i \in \{0, ..\mathit{len}\} \cdot \mathit{head}\textbf{.}[\mathit{next}]^i\textbf{.}\mathit{val}$

spec attr *rep* $=$ $\{\mathit{addr\_head}\} \cup \bigcup i \in \{0, ..\mathit{len}\} \cdot \mathit{head}\textbf{.}[\mathit{next}]^i\textbf{.}\mathit{rep}$

Example: List Implementation

module *ListImpl*

class *Node*

prog attr *val*, *next*

spec attr *inv* $= (\mathit{val} \in \mathbb{Z})$

spec attr *rep* $= \{\mathit{addr\_next}, \mathit{addr\_val}\}$

end class

class *List*

prog attr *head*

spec attr *len* $= \mathit{min}\{i \in \mathbb{N} \cdot \ \mathit{head}.[\mathit{next}]^i = \mathit{null}\}$

spec attr $L = \lambda i \in \{0, ..\mathit{len}\} \cdot \mathit{head}.[\mathit{next}]^i.\mathit{val}$

spec attr *rep* $= \{\mathit{addr\_head}\} \cup \bigcup i \in \{0, ..\mathit{len}\} \cdot \mathit{head}.[\mathit{next}]^i.\mathit{rep}$

spec attr *inv* $=$

$$
\begin{array}{ll}
(& (\forall i \in \{0, ..\mathit{len}\} \cdot \ \mathit{head}.[\mathit{next}]^i \in \mathit{Node} \ \wedge \ \mathit{head}.[\mathit{next}]^i.\mathit{inv}) \\
\wedge & \mathit{disjoint}([\{\mathit{addr\_head}\}]^\frown \lambda i \in \{0, ..\mathit{len}\} \cdot \mathit{head}.[\mathit{next}]^i.\mathit{rep}))
\end{array}
$$

Example: List Implementation

module *ListImpl*

class *Node*

prog attr *val*, *next*

spec attr *inv* $= (val \in \mathbb{Z})$

spec attr *rep* $= \{addr\_next, addr\_val\}$

end class

class *List*

prog attr *head*

spec attr *len* $= min\{i \in \mathbb{N} \cdot head.[next]^i = null\}$

spec attr $L = \lambda i \in \{0, ..len\} \cdot head.[next]^i.val$

spec attr *rep* $= \{addr\_head\} \cup \bigcup i \in \{0, ..len\} \cdot head.[next]^i.rep$

spec attr *inv* $=$

$$(\quad (\forall i \in \{0, ..len\} \cdot head.[next]^i \in Node \wedge head.[next]^i.inv)$$

$$\wedge \quad disjoint([\{addr\_head\}]^\frown \lambda i \in \{0, ..len\} \cdot head.[next]^i.rep))$$

...

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

class *Iterator*

prog attr *attl*

spec attr $pos\ ,\ inv \in \mathsf{Bool}\ ,\ rep$

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

class *Iterator*

prog attr *attl*

spec attr *pos* , *inv* $\in$ Bool , *rep*

$inv \Rightarrow attl \in List \wedge attl.inv \wedge pos \in \{0, ..attl.(\#L) + 1\} \wedge rep \subseteq \mathsf{Dom}\ \sigma$

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

class *Iterator*

prog attr *attl*

spec attr *pos* , *inv* $\in$ Bool , *rep*

$inv \Rightarrow attl \in List \wedge attl.inv \wedge pos \in \{0, ..attl.(\#L) + 1\} \wedge rep \subseteq \mathsf{Dom}\ \sigma$

$inv \Rightarrow disjoint[rep; attl.rep]$

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

class *Iterator*

prog attr *attl*

spec attr *pos* , *inv* $\in$ Bool , *rep*

$inv \;\Rightarrow\; attl \in List \;\wedge\; attl.inv \;\wedge pos \in \{0, ..attl.(\#L) + 1\} \;\wedge\; rep \subseteq \mathsf{Dom}\;\sigma$

$inv \;\Rightarrow\; disjoint[rep; attl.rep]$

$inv \;\Rightarrow\; rep\ \mathbf{frames}\ (attl, rep) \;\wedge\; (rep \cup attl.rep)\ \mathbf{frames}\ (inv, pos)$

Example: Iterators Specification

module *IteratorSpec*

import *ListSpec*

class *Iterator*

prog attr *attl*

spec attr pos , $inv \in \mathsf{Bool}$, rep

$inv \quad \Rightarrow \quad attl \in List \ \wedge \ attl\mathbf{.}inv \ \wedge pos \in \{0, ..attl\mathbf{.}(\#L) + 1\} \ \wedge \ rep \subseteq \mathsf{Dom}\ \sigma$

$inv \quad \Rightarrow \quad disjoint[rep; attl\mathbf{.}rep]$

$inv \quad \Rightarrow \quad rep\ \mathbf{frames}\ (attl, rep) \ \wedge \ (rep \cup attl\mathbf{.}rep)\ \mathbf{frames}\ (inv, pos)$

method $next() \cdot$

$$inv \ \wedge \ pos < attl\mathbf{.}(\#L)$$

$$\Rightarrow \quad \Delta rep \ \wedge \ inv' \ \wedge \ pos' = pos + 1 \ \wedge \ attl' = attl \ \wedge \ rep' \subseteq rep \cup Unused$$

end class

end module

Example: Iterators Implementation

module *IteratorImpl*

import *ListImpl*

class *Iterator*

prog attr *attl* , *currentNode*

Example: Iterators Implementation

module *IteratorImpl*

import *ListImpl*

class *Iterator*

prog attr *attl* , *currentNode*

spec attr $pos \;=\; min\{i \in \mathbb{N} \cdot \; attl\mathbf{.}head\mathbf{.}[next]^i = currentNode\}$

Example: Iterators Implementation

module *IteratorImpl*

import *ListImpl*

class *Iterator*

prog attr *attl* , *currentNode*

spec attr $pos \ = \ min\{i \in \mathbb{N} \cdot \ attl.head.[next]^i = currentNode\}$

spec attr $rep = \{addr\_attl \ , \ addr\_currentNode\}$

Example: Iterators Implementation

module *IteratorImpl*

import *ListImpl*

class *Iterator*

prog attr *attl* , *currentNode*

spec attr *pos* $= min\{i \in \mathbb{N} \cdot attl.head.[next]^i = currentNode\}$

spec attr *rep* $= \{addr\_attl , addr\_currentNode\}$

spec attr *inv* $= (\quad attl \in List \wedge attl.inv \wedge pos \in \{0, ..attl.(\#L) + 1\}$

$\quad \wedge \quad rep \subseteq \mathsf{Dom}\, \sigma \wedge disjoint[rep ; attl.rep])$

Example: Iterators Implementation

module *IteratorImpl*

import *ListImpl*

class *Iterator*

prog attr *attl* , *currentNode*

spec attr $pos \; = \; min\{i \in \mathbb{N} \cdot \; attl.head.[next]^i = currentNode\}$

spec attr $rep = \{addr\_attl \; , \; addr\_currentNode\}$

$$\textbf{spec attr } inv \; = \; (\quad attl \in List \; \wedge \; attl.inv \; \wedge \; pos \in \{0, ..attl.(\#L) + 1\}$$
$$\wedge \quad rep \subseteq \mathsf{Dom}\, \sigma \; \wedge \; disjoint[rep \; ; \; attl.rep])$$

method $next() \cdot \; currentNode := currentNode.next$

end class

end module

Related Work

- (Leino, Nelson 2002): too complex and restrictive

Related Work

- (Leino, Nelson 2002): too complex and restrictive
- Universes (Müller 2002)
  - based on *ownership*

Related Work

- (Leino, Nelson 2002): too complex and restrictive
- Universes (Müller 2002)
  - based on *ownership*
  - *ownership transfer restriction*: objects do not cross encapsulation borders

Related Work

- (Leino, Nelson 2002): too complex and restrictive
- Universes (Müller 2002)
  - based on *ownership*
  - *ownership transfer restriction*: objects do not cross encapsulation borders
  - *sharing visibility restriction*: a shareable resource must know all its clients

Related Work

- (Leino, Nelson 2002): too complex and restrictive
- Universes (Müller 2002)
  - based on *ownership*
  - *ownership transfer restriction*: objects do not cross encapsulation borders
  - *sharing visibility restriction*: a shareable resource must know all its clients
- Boogie (Leino, Müller 2004)
  - removes ownership transfer restriction

Related Work

- (Leino, Nelson 2002): too complex and restrictive
- Universes (Müller 2002)
  - based on *ownership*
  - *ownership transfer restriction*: objects do not cross encapsulation borders
  - *sharing visibility restriction*: a shareable resource must know all its clients
- Boogie (Leino, Müller 2004)
  - removes ownership transfer restriction
- Separation Logic (O'Hearn et al. 2001, 2004), (Parkinson and Biermann 2005)
  - non-standard logic
  - no support for sharing

Conclusion

Simplicity of Dynamic Frames

- Special case of specification attributes

Conclusion

Simplicity of Dynamic Frames

- Special case of specification attributes

Flexibility of Dynamic frames

- Objects may cross encapsulation boundaries

Conclusion

Simplicity of Dynamic Frames

- Special case of specification attributes

Flexibility of Dynamic frames

- Objects may cross encapsulation boundaries
- Sharing is supported

Conclusion

Simplicity of Dynamic Frames

- Special case of specification attributes

Flexibility of Dynamic frames

- Objects may cross encapsulation boundaries
- Sharing is supported
- Classes do not need to know sharing clients

Conclusion

Simplicity of Dynamic Frames

- Special case of specification attributes

Flexibility of Dynamic frames

- Objects may cross encapsulation boundaries
- Sharing is supported
- Classes do not need to know sharing clients

Part of author's PhD thesis:

A Theory of Object Oriented Refinement
(University of Toronto 2006)

available at:

`http://www.cs.toronto.edu/~hehner/aToOOR.pdf`