Dynamic Frames
ICTAC Summer School

Jim Woodcock

University of York

August 2011
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Pointers in Unifying Theories of Programming

- objectives for this part of the project:
  1. choose a pointer model for FreeRTOS refinement
  2. develop idealised code for priority queue
  3. compare with existing code
  4. reconstruct rational development of existing code

- candidate theories for pointer model
  1. naive pointer model
  2. abstract sharing model
  3. dynamic frames
  4. separation logic
Abstract sharing model

- Cavalcanti, Harwood, & Woodcock
- recursive record type $List = (val : \mathbb{Z}; next : List)$
- objectives
  1. describe sharing amongst references
  2. remain abstract from actual storage model
- three alphabet variables:
  - $A$: a set of hierarchical addresses
    - all legal addresses that could be constructed
  - $V$: partial function from addresses to values
    - maps addresses of primitive attributes to their values
  - $S$: equivalence relation on addresses
    - relates addresses that share a common location
- $A \setminus \text{dom } V$: acceptable addresses that yield objects
Key idea

- assume all object and primitive values have a location
- variables and attribute accesses are names of locations
- distinguishing feature of model
  - language recogniser for set of legal addresses
  - updatable automaton relating addresses to values
- cf. Hoare & He’s traces model
- Paige’s bunches model
- simple example:

\[
A = (l.\text{next}^*.\text{val}) \leq \\
V = l.\text{next}^*.\text{val} \times \{1\} \\
S = \{\{l.\text{next}^*\}, \{l.\text{next}^*.\text{val}\}\}
\]
Unifying Theories of Programming

- alphabet
  - observations of interest for a theory
- syntax
  - for denoting elements of the theory
- healthiness conditions
  - for membership of the theory
- UTP designs:
  - subset of basic relational programming model
  - pre- and postconditions
- make sharing a subtheory of designs
Healthiness conditions

- addresses in $A$ are prefix closed

$$FAd : \text{finite Addresses}$$

$$\text{HP1}(P) = \forall a_1 : FAd; a_2 : A \mid a_1 \leq a_2 \bullet a_1 \in A$$

- $V$-addresses are finite and terminal

$$\text{term}(A) = \{ x : X \cap FAd \mid \neg \exists y : X \bullet x < y \}$$

$$\text{HP2}(P) = (\text{dom } V = \text{term}(A))$$
Healthiness conditions

- \( \prime \): name of variable \( v \)
- programming variables have values assigned by automaton

\[
\text{vars}(X) \triangleq \{ x : X \, \bullet \, x(1) \}
\]

\[
\text{HP3}(P) = (v = V(\prime v)) \land (\text{vars}(A) = v)
\]

- \( A \) is an equivalence relation

\[
\text{HP4}(P) = S \in \text{EquivRel}(A)
\]
Healthiness conditions

- forward closure of addresses
- sharing is forward closed

\[ \text{HP5}(P) = \forall x, y, a : Ad \]
\[ (x, y) \in S \land (x.a \in A \lor y.a \in A) \]
\[ \bullet (x.a, y.a) \in S \]

- terminals sharing location must have same value

\[ \text{HP6}(P) = \forall a, b : Ad \]
\[ (a, b) \in S \land a \in \text{dom } V \]
\[ \bullet b \in \text{dom } V \land (V(a) = V(b)) \]
Results for abstract sharing model

- elegant model, but many healthiness conditions (12!)
- proved closure results
  - UTP designs
  - simple programming language
    - value and pointer assignment, nondeterministic choice, conditional, sequential composition, recursion

- in use in hiJaC project for Safety-Critical Java
- still a holistic model
- lacks modular reasoning technique
- can we reason solely in terms of contractual interface?
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Dynamic frames

- functional and framing requirements

\[
C = x' = x + 1
\]

\[
\ldots y := 0; \ C
\]

- non-modular setting: \( x' = x + 1 \land y' = y \land z' = z \)
- modular setting: \textbf{ensures} \( x' = x + 1 \) \textbf{modifies} \( x \)
- encapsulation
  - client doesn’t even know implementation variables
- ghost variable, pure methods
Abstract aliasing

module ASpec
    spec var $S : \mathbb{P} \mathbb{Z}$
    insert($x : \mathbb{Z}$) ensures $S' = S \cup \{x\}$
    find($x : \mathbb{Z}$) ensures $S' = S \land \text{return}' = (x \in S)$
end module

module ARef
    prog var $L : \mathbb{Z}^*$
    spec var $S = \{ x : \mathbb{Z} | \exists i : \mathbb{N} \bullet i < \#L \land x = L(i) \}$
    insert($x : \mathbb{Z}$) ensures $L' = \langle x \rangle \cap L$
    find($x : \mathbb{Z}$) ensures $L' = L$
    \quad \land \text{return}' = (\exists i : \mathbb{N} \bullet i < \#L \land x = L(i))$
end module
Abstract aliasing

- framing requirement for \textit{insert}: \textbf{modifies} $S$
- translated to $y' = y \land z' = z$ in wider context
- translation unsound for pointers
- what if representation of $y$ shares heap locations with representation of $S$?
- abstract aliasing problem
- solutions:
  - impose restrictions to avoid problem
  - make abstract aliasing directly expressible
- dynamic frame theory is just a specification pattern
- semantics for separation logic?
Observations

- infinite set of locations $Loc$
- infinite set of object references: $O$
- null reference: $null \notin O$
- infinite set of values $Val = B \cup Z \cup O$
- region: any subset of $Loc$
- $\Sigma = Loc \rightarrow Val$
- store $\sigma : \Sigma$
- store expressions: $\alpha e = \{\sigma\}$
- store relations: $\alpha P = \{\sigma, \sigma'\}$
- used locations $Used = \text{dom} \sigma$
- $Unused = Loc \setminus Used$
Program constructs

- concrete assignment:

  \[ x := E = (\sigma := \sigma \oplus \{ \text{addr}_x \mapsto E\}) \]

- pointer assignment

  \[ l : \text{location-valued expression} \]

  \[ *l := E = \sigma := \sigma \oplus \{ l \mapsto E\} \]

- local program variable introduction (fresh \(x\)):

  \[ \text{var } x \bullet P = \exists \text{addr}_x : \text{Unused} \bullet P \]
Frame preservation

- region: set of locations
- preserves operator: $\Xi$
  - what must be left untouched (for region $f$):
    $$\Xi f = (f \triangleleft \sigma' = f \triangleleft \sigma)$$
- antimonotonicity: $f \subseteq g \Rightarrow (\Xi g \Rightarrow \Xi f)$

Proof

$$\Xi g$$
$$= (g \triangleleft \sigma' = g \triangleleft \sigma)$$
$$\Rightarrow (f \triangleleft (g \triangleleft \sigma')) = f \triangleleft (g \triangleleft \sigma))$$
$$= ((f \cap g) \triangleleft \sigma' = (f \cap g) \triangleleft \sigma)$$
$$= (f \triangleleft \sigma' = f \triangleleft \sigma)$$
$$= \Xi f$$
Monotonicity of preservation

- modifies operator: $\Delta$
- what may be changed:

$$\Delta f = \Xi(Used \setminus f)$$

- monotonicity:

$$f \subseteq g \Rightarrow (\Delta f \Rightarrow \Delta g)$$

note: $Used \setminus g \subseteq Used \setminus f$

Proof

$$\Delta f$$

$$= \Xi(Used \setminus f)$$

$$\Rightarrow \Xi(Used \setminus g)$$

$$= \Delta g$$
Expression framing

- Let $f$ be a region and $E$ an expression on $\sigma$.
- $f$ frames $E$ in state $\sigma$ if the following holds:

\[
 f \text{ frames } E = \forall \sigma' : \Sigma \bullet \exists f \Rightarrow (E' = E)
\]

- $E$, $D$ on $\sigma$ are independent in $\sigma$ if

\[
 f \text{ frames } E \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle
\]
Theorem: Value Preservation

let $f, g$ be regions and $D$ be an expression on $\sigma$

$\Delta f \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle \Rightarrow (D' = D)$

Proof

$\Delta f \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle$

$= \Xi(\text{Used } \backslash f) \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle$

$\Rightarrow \Xi((\text{Used } \backslash f) \cap g) \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle$

$= \Xi((\text{Used } \cap g) \backslash f) \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle$

$= \Xi(g \backslash f) \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle$

$\Rightarrow \Xi g \land g \text{ frames } D$

$\Rightarrow (D' = D)$
Dynamic Frames and Variable Framing

- **dynamic frame:**
  - specification variable $f$ at state $\sigma$ with $f \subseteq Used$

- introduce frame $f$ for every specification variable $v$

- invariant: $inv \Rightarrow f \subseteq Used$

- specification variable framing property for $v$:
  $inv \Rightarrow f \text{ frames } v$

- typically, $rep$: the representation region

- program variables don’t need frames
  - frame for program variable $m$ is $\{addr\_m\}$
Dynamic frames method

- implementor’s obligations
  - find implementations for specification attributes...
  - find implementations for frames...
  - ...that satisfy theory requirements

- value preservation theorem:

\[ \Delta f \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle \implies (D' = D) \]

- implementor’s responsibility:
  - ensure \( \Delta f \), without knowing about \( g \) and \( D \)

- client’s responsibility:
  - ensure disjoint \( \langle f, g \rangle \) as well as \( g \text{ frames } D \)
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Attributes and Methods

- current object: \( self \in \mathcal{O} \)
- specification attribute
  - specification attributes abstract hidden state
  - state expression: \( \alpha e = \{\sigma, self\} \)
- program attribute
  - special case of specification attribute
  - content of state at location \( addr_x \)
  - \( x = \sigma(addr_x) \)
  - \( addr_x \) depends only on \( self \)
  - axiom: \( (p.q)[E/x] = p[E/x].q[E/x] \)
  - for any \( k \in \mathbb{N} \)

\[
[E]^0 = self \\
[E]^{k+1} = [E]^k.E
\]
Stereotypical specification attributes

- for each object: 3 specification attributes
  - initialization constraint \( init \)
  - invariant \( inv \)
  - representation region \( rep \)

- consistency, for all object references and states

\[
\begin{align*}
\text{init} & \in \mathbb{B} \\
\land \text{inv} & \in \mathbb{B} \\
\land (\text{init} & \Rightarrow \text{inv}) \\
\land (\text{inv} & \Rightarrow \text{rep} \subseteq \text{Used}) \\
\land \text{null} \cdot \text{rep} & = \emptyset
\end{align*}
\]

- method invocation
  - for object reference \( o \), identifier \( l \), and values \( x, y, \ldots \)

\[
o \cdot l(x, y, \ldots)
\]
Class specifications

- class: set of object references
- axioms: $\textit{self}, \sigma, \sigma'$ implicitly universally quantified
- attribute specifications: $a = E$
- method specifications:

  \[
  \text{method } l(x; y; \ldots) \bullet S = \\
  (\forall x, y, \ldots \bullet \textit{self}.l(x; y; \ldots) \Rightarrow S)
  \]

- object creation:

  \[
  x := \textbf{new } C = \Delta \{\text{addr}_x\} \\
  \land x' \in C \\
  \land (x.\text{init})' \\
  \land (x.\text{rep})' \subseteq \text{Unused} \setminus \{\text{addr}_x\}
  \]
Class specifications

```java
class Node
    prog attr val, next
    init = val ∈ ℤ ∧ (next = null)
    inv = val ∈ ℤ ∧ (next ≠ null ⇒ next ∈ Node)
    rep = {addr_val, addr_next}
end class
```

Notation

- `next` is an abbreviation for `self.next`
- `...` which is an abbreviation for `self.σ(addr_next)`
- `next'` is an abbreviation for `self.next'`
- `...` which is an abbreviation for `self.σ'(addr_next)`
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Preserving disjointness

- suppose we know that \( f \) and \( g \) are disjoint
- suppose we update \( f \) by extending it with unused locations
- how do we prove \( f \) and \( g \) remain disjoint?
- recall Value Preservation Theorem

\[
\Delta f \land g \text{ frames } D \land \text{disjoint } \langle f, g \rangle \Rightarrow (D' = D)
\]

- take \( D = g \) to use this to prove that \( g \) doesn’t change
- \( g \) frames \( g \): self-framing dynamic frame
Theorem: Disjointness Preservation

\[ f \cup g \subseteq \text{Used} \]
\[ \land g \text{ frames } g \]
\[ \land \text{disjoint } \langle f, g \rangle \]
\[ \land \Delta f \land f' \setminus f \subseteq \text{Unused} \]
\[ \Rightarrow (\text{disjoint } \langle f, g \rangle)' \]

Proof

\[ f \cup g \subseteq \text{Used} \land g \text{ frames } g \land \text{disjoint } \langle f, g \rangle \land \Delta f \land f' \subseteq f \cup \text{Unused} \]
\[ \Rightarrow f \cup g \subseteq \text{Used} \land \text{disjoint } \langle f, g \rangle \land f' \subseteq f \cup \text{Unused} \land (g' = g) \]
\[ \Rightarrow f \cup g' \subseteq \text{Used} \land \text{disjoint } \langle f, g' \rangle \land f' \subseteq f \cup \text{Unused} \]
\[ = \text{disjoint } \langle f, g', \text{Unused} \rangle \land f' \subseteq f \cup \text{Unused} \]
\[ \Rightarrow \text{disjoint } \langle f \cup \text{Unused}, g' \rangle \land f' \subseteq f \cup \text{Unused} \]
\[ \Rightarrow \text{disjoint } \langle f', g' \rangle \]
Auxiliary notation

abstract assignment

▶ suppose $inv \Rightarrow rep$ frames $(inv, rep, x, y)$

\[
x : \overset{=} E = \\begin{align*}
inv & \Rightarrow x' = E \land y' = y \land inv' \\
& \land \Delta rep \land rep' \subseteq rep \cup Unused
\end{align*}
\]

strong framing

▶ suppose $P$ satisfies $\Delta f$ where $f$ is self-framing
▶ then $P \Rightarrow f' \subseteq f \cup Unused$
▶ strong frame

\[
\overline{\Delta f} = \Delta f \land f' \subseteq f \cup Unused
\]
Lemma: Dash distribution

\[(p.q)' = p'.q'\]

Proof

\[(p.q)'
\= (p.q)[\sigma'/\sigma]
\= p[\sigma'/\sigma].q[\sigma'/\sigma]
\= p'.q'\]
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Module: List specification

module ListSpec

class List

spec attr L

inv ⇒ L ∈ ℤ* ∧ rep ⊆ Used ∧ rep frames (rep, inv, L)
init ⇒ L = ⟨⟩ ∧ inv

method push(x) • x ∈ ℤ ⇒ L := ⟨x⟩ ∩ L

method pop(l) •

inv ∧ l ∈ Loc \ rep ∧ L ≠ ⟨⟩
⇒ \Δ(\{l\} ∪ rep) ∧ L' = tail(L) ∧ inv'
∧ l ∈ Used ∧ σ'(l) = head(L) ∧ σ'(l).inv
∧ σ'(l).rep ⊆ rep ∪ Unused
∧ disjoint ⟨rep', σ'(l).rep, \{l\}⟩

end class

end module
module ListImpl

class Node

    prog attr val, next

    init = val ∈ ℤ ∧ (next = null)
    inv = val ∈ ℤ ∧ (next ≠ null ⇒ next ∈ Node)
    rep = {addr_val, addr_next}

end class

class List

    ...

end class

end module
class List
    prog attr list
    spec attr \( \text{len} = \min \{ i : \mathbb{N} \mid \text{list.[next]}^i = \text{null} \} \)
    spec attr \( L = \langle i : 0 \ldots \text{len} - 1 \mid \text{list.[next]}^i . \text{val} \rangle \)
    rep = \{ addr\_list \} \cup \bigcup \{ i : 0 \ldots \text{len} - 1 \mid \text{list.[next]}^i . \text{rep} \}
    inv =
        rep \subseteq \text{Used} \\
        \land \text{disjoint} (\langle \{ \text{addr\_list} \} \rangle \cap \langle i : 0 \ldots \text{len} - 1 \mid \text{list.[next]}^i . \text{rep} \rangle)
    init = (\text{list} = \text{null})
    method push(x) •
        var n •
            n := \text{new Node}; (n.val, n.next, list := x, list, n)
    method pop(l) •
        \text{len} \neq 0 \land \text{inv} \Rightarrow \ast l := \text{list.val}; \text{list} := \text{list.next}
Correctness

- module $B$ implements module $A$ iff
  - the names declared in $A$ are all included in $B$
  - the axioms of $B$ imply the axioms of $A$

- specification attributes
  1. invariant

  \[ L \in \mathbb{Z}^* \land rep \subseteq Used \land rep \text{ frames } (rep, inv, L) \]

  2. initialisation

  \[(list = null) \Rightarrow L = \langle \rangle \land inv\]

- straightforward refinement proof...

- ... except for $rep \text{ frames } (rep, inv, L)$
Theorem: Chain framing

\{i : 0 \ldots k - 1 \Rightarrow [a]^i \cdot addr_a\} frames [a]^k

Proof: by induction

Basis: \(k = 0\)

\{i : 0 \ldots 0 - 1 \Rightarrow [a]^i \cdot addr_a\} frames [a]^0

= \emptyset frames self
Inductive step

Assume true for \( k = j \)

\[
\{ i : 0 \ldots j - 1 \bullet [a]^i.\text{addr}_a \} \text{ frames } [a]^j
\]
\[
= \forall \sigma' : \Sigma \bullet \exists \{ i : 0 \ldots j - 1 \bullet [a]^i.\text{addr}_a \} \Rightarrow ([a]^j)' = [a]^j
\]

Proof of inductive step \( k = j + 1 \):

\[
\exists \{ i : 0 \ldots j \bullet [a]^i.\text{addr}_a \}
\]
\[
= \exists \{ i : 0 \ldots j - 1 \bullet [a]^i.\text{addr}_a \} \land [a]^j.\sigma'(\text{addr}_a) = [a]^j.\sigma(\text{addr}_a)
\]
\[
\Rightarrow ([a]^j)' = [a]^j \land [a]^j.\sigma'(\text{addr}_a) = [a]^j.\sigma(\text{addr}_a)
\]
\[
= ([a]^j)' = [a]^j \land [a]^j.a' = [a]^j.a
\]
\[
\Rightarrow ([a]^j)' . a' = [a]^j . a
\]
\[
= ([a]^j . a)' = [a]^j . a
\]
Lemma

\[ \Xi \text{rep} = \Xi (\{ \text{addr\_list} \} \cup \bigcup \{ j : 0 \ldots \text{len} - 1 \bullet \text{list.}[\text{next}]^j . \text{rep} \} ) \]
\[ \Rightarrow \Xi (\bigcup \{ j : 0 \ldots \text{i} - 1 \bullet \text{list.}[\text{next}]^j . \{ \text{addr\_val}, \text{addr\_next} \} \} ) \]
\[ \Rightarrow \Xi \{ j : 0 \ldots \text{i} - 1 \bullet \text{list.}[\text{next}]^j . \text{addr\_next} \} \]  \[1\]

chain framing

\{ j : 0 \ldots \text{i} - 1 \bullet [\text{next}]^j . \text{addr\_next} \} \textbf{frames} [\text{next}]^i
\[ \Rightarrow \{ j : 0 \ldots \text{i} - 1 \bullet \text{list.}[\text{next}]^j . \text{addr\_next} \} \textbf{frames} \text{list.}[\text{next}]^i \]  \[2\]

therefore

\((\text{list.}[\text{next}]^i)' = \text{list.}[\text{next}]^i\)
Theorem: \( \text{rep frames } \text{len} \)

\[ \text{len}' = \min \{ i : \mathbb{N} \mid (\text{list}.[\text{next}]^i)' = \text{null} \} \]

\[ = \min \{ \min \{ i : \mathbb{N} \mid i \leq \text{len} \cdot (\text{list}.[\text{next}]^i)' = \text{null} \}, \]
\[ \min \{ i : \mathbb{N} \mid i > \text{len} \cdot (\text{list}.[\text{next}]^i)' = \text{null} \} \} \]

\[ = \min \{ \min \{ i : \mathbb{N} \mid i \leq \text{len} \cdot \text{list}.[\text{next}]^i = \text{null} \}, \]
\[ \min \{ i : \mathbb{N} \mid i > \text{len} \cdot (\text{list}.[\text{next}]^i)' = \text{null} \} \} \]

\[ = \min \{ \text{len}, \min \{ i : \mathbb{N} \mid i > \text{len} \cdot (\text{list}.[\text{next}]^i)' = \text{null} \} \} \]

\[ = \text{len} \]
Lemma: \textbf{rep frames } \textbf{L}

note that

\[\text{list.[next]}^i.\text{addr_val} \in \text{rep}\]

assume \( \overline{\text{rep}} \)

\[L'\]
\[= \langle i : 0 \ldots \text{len'} - 1 \bullet (\text{list.[next]}^i.\text{val}') \rangle\]
\[= \langle i : 0 \ldots \text{len} - 1 \bullet (\text{list.[next]}^i.\text{val}') \rangle\]
\[= \langle i : 0 \ldots \text{len} - 1 \bullet (\text{list.[next]}^i)' .\text{val}' \rangle\]
\[= \langle i : 0 \ldots \text{len} - 1 \bullet \text{list.[next]}^i .\text{val}' \rangle\]
\[= \langle i : 0 \ldots \text{len} - 1 \bullet \text{list.[next]}^i .\text{val} \rangle\]
\[= L\]
Remainder of correctness proof

- $rep$ frames ($rep$, $inv$): straightforward
- initialisation: straightforward
- $push$, $pop$: from program construct semantics

Conclusions

- verbose specifications
- patterns and stereotypic refinements will help
- reasoning about dynamic frames is straightforward
- reasoning can be streamlined
  - library of theorems like Chain Framing
Outline

Pointers in Unifying Theories of Programming

Dynamic frames

Simple object model

Frame Disjointness

Example

Next steps
Next steps

- use dynamic frame theory in FreeRTOS refinement
- verify the priority queues (doubly linked lists)
- embed dynamic-frame-designs in concurrency theory
- use this in multi-core work

Acknowledgements

- Leo Freitas
- Yannis Kassios
- Ana Cavalcanti & Andy Wellings
- Ian Sommerville & INDEED project