Regions and Permissions for Data Invariants

Romain Bardou and Claude Marché

Septembre 2009
Motivation

preservation of data invariants in pointer programs
  ▶ ownership system of Spec# [Barnett et al 04]

static typing instead of theorem provers
  ▶ Universe Types [Dietl, Müller 05]

how?
  ▶ regions [Tofte, Talpin, Jouvelot 91] ... [Banerjee et al 08]
  ▶ with permissions [Crary et al 99]
class PosInt {
    int value;
    //@ invariant this.value > 0;

    void double() {
        value := value + value;
    }
}
Core Language

functional style with references \((e_1 := e_2, !e)\)

\[
\text{type } \text{PosInt} = \\
\quad \text{int} \\
\quad \text{inv}(\text{this}) = !\text{this} > 0 \\
\text{end}
\]

\[
\text{val } \text{double}(x: \text{PosInt}): \text{unit} = \\
\quad x := !x + !x
\]

focus on pointers and aliasing
ignore inheritance and dynamic dispatch
Problem: Pointer Aliasing

```haskell
val f(x: PosInt, y: PosInt): unit =
    x := 0;
    x := 1 / !y

what if x = y?
```

![Diagram](image-url)
Problem: Components

```
type SortedPair =
  PosInt × PosInt
inv(this) = !this.1 < !this.2
end

val double(x: PosInt): unit =
x := !x + !x
```

what if \( x \) is member of a SortedPair \( p \)?
Regions

solution: group pointers by regions

pointers of two different regions may not be aliased
permission = static linear information about a region

“linear” means:
- permissions cannot be duplicated
- permissions depend on the program point
- operations may consume some permissions
- operations may produce other permissions
Empty Regions

regions are created empty

\textbf{region } \rho \textbf{ in}

this produces permission \(\rho^\emptyset\): “\(\rho\) is empty”
Allocation and Singleton Regions

Pointers are allocated in empty regions

```
new PosInt[ρ]
```

This:

- consumes permission $\rho^\emptyset$
- produces permission $\rho^S$: “$\rho$ is singleton”

Region $\rho$ is no longer empty: it is singleton
a singleton region $\rho$ may be demoted to a group region

this is implicit

this:

- consumes permission $\rho^S$
- produces permission $\rho^G$: “$\rho$ is group”
Adoption

adoption moves a pointer from a singleton region to an already-existing group region

if $x$ is in region $\sigma$:

```
adopt $x$ in $\rho$
```

this:

- consumes permissions $\sigma^S$ and $\rho^G$
- produces permission $\rho^G$
The Permission Diagram (so far)
Permissions for Invariants

use permissions to denote whether invariants hold

- $\rho^\emptyset$: empty region, no invariant
- $\rho^\circ$: open singleton region, invariant does not hold
- $\rho^\times$: closed singleton region, invariant holds
- $\rho^G$: group region, all invariants hold

only pointers in open regions can be assigned
Packing and Unpacking

**pack**

packing a pointer of $\rho$:
- consumes $\rho^\circ$
- produces $\rho^\times$
- generates a proof obligation (the invariant)

**unpack**

unpacking is the opposite operation:
- consumes $\rho^\times$
- produces $\rho^\circ$
The Permission Diagram (with packing)

\[ \sigma^\emptyset \xrightarrow{\text{new}} \sigma^0 \xrightarrow{\sigma \times} \rho^G \xrightarrow{\text{adopt}} \sigma^G \]
Owned Regions

problem: invariants about other pointers?

```plaintext
type SortedPair ⟨ρ₁, ρ₂⟩ = PosInt[ρ₁] × PosInt[ρ₂]
inv(this) = !this.1 < !this.2
end

val bad(x: SortedPair⟨ρ₁, ρ₂⟩[ρ])
  consumes ρₓ, ρ₁°, ρ₂°
  produces ρₓ, ρ₁°, ρ₂° =
    !x.1 := 69;
    !x.2 := 42
```

ρ₁

ρ₂

ρ

!x.1

!x.2
Owned Regions

solution: owned regions

type SortedPair =
  own ρ₁, ρ₂
  PosInt[ρ₁] × PosInt[ρ₂]
inv(this) = !this.1 < !this.2
end
The Permission Diagram (with owned regions)

\[ \sigma \times \sigma \]

\[ \rho \]

\[ \text{own}(\sigma)^G \]

\[ \text{pack} \]

\[ \text{unpack} \]

\[ \text{adopt} \]

Regions and Permissions for Data Invariants
Group to Singleton?

problem: how to modify a pointer of a group region?
Group to Singleton?

solution: **extract** the pointer to a singleton region

problem: what happens to the group region?
  - what if several pointers are extracted?
  - what if a pointer is extracted several times?

solution: group region **temporarily disabled**
Linear Implication

\[ \sigma \circ \rho \]

\( \rho \) is disabled temporarily

\( \sigma^\times \) must be given to enable \( \rho \)

allows temporary extraction from \( \rho \) to \( \sigma \)
if $y$ in region $\rho$:

**focus $y$ in $\sigma$**

this:

- consumes $\sigma^\emptyset$ and $\rho^G$
- produces $\sigma^\times$ and $\sigma \rightarrow^\circ \rho$

region $\sigma$ now also contains $y$
Unfocus

if \( y \) in region \( \sigma \):

\[ \mathbf{unfocus} \ y \ \mathbf{in} \ \rho \]

this:
- consumes \( \sigma^\times \) and \( \sigma \rightarrow \rho \)
- produces \( \rho^G \)

region \( \sigma \) is disabled definitely
Focus and Unfocus Usage

if \( x \) in group region \( \rho \):

\[
\begin{align*}
\text{region } \sigma \text{ in} \\
\text{let } x_f = (\text{focus } x \text{ in } \sigma) \text{ in} \\
\text{unpack } x_f; \\
x_f := \cdots; \\
\text{pack } x_f; \\
\text{unfocus } x_f \text{ in } \rho
\end{align*}
\]

\( x = x_f \), but:

- \( x \) is in \( \rho \)
- \( x_f \) is in \( \sigma \)
Soundness

Definition
heap is coherent w.r.t. \( \bar{\Sigma} \):
  - invariants of closed pointers hold
  - ...

Theorem
If:
  - \( e \) is well-typed w.r.t. types, regions, permissions
    - when given permissions \( \bar{\Sigma} \), \( e \) gives back \( \bar{\Sigma}' \)
  - \( e \) and heap \( \mathcal{H} \) reduce to \( e' \) and \( \mathcal{H}' \)
  - \( \mathcal{H} \) is coherent w.r.t. \( \bar{\Sigma} \)
then:
  - \( \mathcal{H}' \) is coherent w.r.t. \( \bar{\Sigma}' \)
Conclusion

**static** type system with regions and permissions

guarantees **invariant preservation**

- only VCs: invariants, when packing

**ownership** at the level of regions

can handle examples such as observer pattern

can handle some form of **abstraction**

- owned regions can be hidden
Need for Inference

inference of region annotations

val f(): PosInt[ρ] =
  region σ in
  let x = new PosInt[σ] in
  x := 5;
  pack x;
  let x = (adopt x in ρ) in
  region σy in
  let y = (focus x in σy) in
  unpack y;
  y := 7;
  pack y;
  unfocus y in ρ;
  y

val f(): PosInt =
  let x = new PosInt in
  x := 5;
  x := 7;
  x
Future Works

more powerful abstraction using refinement approaches

inference

▶ current direction: given function prototypes and focus annotations, infer remaining annotations