Université de Paris-Sud 11
Centre d'Orsay

INRIA Saclay – Île-de-France
Equipe ProVal

Vérification de programmes avec pointeurs à l'aide de régions et de permissions

Romain Bardou

présentée le 14 octobre 2011 devant le jury composé de :

MM. Peter Müller
François Pottier
Jean Goubault-Larrecq
Burkhart Wolff
Claude Marché

There are two ways to write error-free programs;
only the third one works.

— Alan J. Perlis

Program Verification

programming needs thinking
verification is tedious

	human	machine
thinking	good	bad
repetition	bad	good

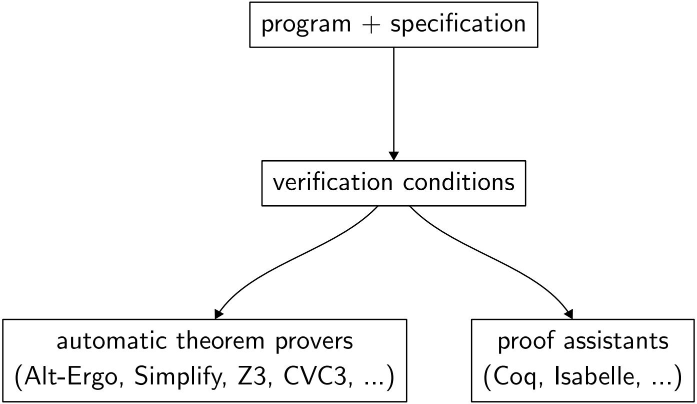
parts of verification are repetitive
$\Longrightarrow$ let the human program and the machine verify

Trade-Off: Automation vs. Expressiveness

properties:

“x is always an integer”	automated (typing)
“x is always an odd integer”	requires reasoning (annotations)
“for all i, $a[i]$ is prime”	requires more reasoning (proofs)

Deductive Program Verification



program + specification

verification conditions

automatic theorem provers
(Alt-Ergo, Simplify, Z3, CVC3, ...)

proof assistants
(Coq, Isabelle, ...)

Deductive Program Verification

expressiveness:

- mainstream programming languages (C, Java...)
- (at least) first-order logic for specifications

automation:

- specification written by hand
- automatic provers for simple verification conditions
- proof assistants for difficult verification conditions

Deductive Verification: Example

```
void max(int i, int j)
  /*@ ensures \result >= i && \result >= j */
{
  if (i > j)
    return i;
  else
    return j;
}
```

verification conditions:

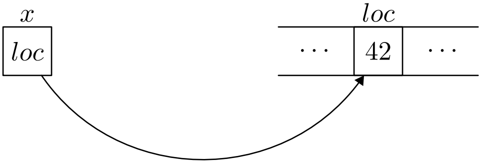
$$i > j \Rightarrow i \geq i \land i \geq j$$

$$\neg(i > j) \Rightarrow j \geq i \land j \geq j$$

Pointers

pointer = variable containing a location

pointed value = value stored at location



Pointer Aliasing

```
*p = 42;
*q = 69;
/*@ assert *p = 42; */
```

what if $p = q$?

verification conditions?

Data Invariants: Examples

handy specification tool

“this array is always sorted”

“this tree is a search tree”

“this tree is well-balanced”

“rocket speed is always positive”

Related Work

ownership

- Data Groups [Leino 1998]
- Ownership Types [Clarke, Potter, Noble 1998]
- Spec# Methodology [Barnett et al. 2004]
- Universe Types [Dietl, Muller 2005]
- Considerate Reasoning [Summers, Drossopoulou 2010]

alias control

- Separation Logic [Reynolds 2002]
- Regional Logic [Banerjee, Naumann, Rosenberg 2008]
- (implicit) Dynamic Frames [Kassios 2006; Smans et al. 2009]

Regions, Permissions / Capabilities, Alias Types...

Main Contribution

A type system using regions and permissions to structure the heap in a modular fashion, control pointer aliasing and data invariants and produce proof obligations where pointers are separated.

implemented as a tool called **Capucine**

Contents

Classes

class = record + invariant + owned regions

```
class Pair
{
  fst: int;
  snd: int;
  invariant fst < snd;
}
```

Pointer Types and Regions

region = set of locations
memory structured using regions

the type of a pointer $[\rho]$ gives its region ρ

```
fun incrPair [r: Pair] (p: [r]): unit
{
  p.fst ← p.fst + 1;
  p.snd ← p.snd + 1;
}
```

Life Cycle of Pointers

- allocation
- initialization of fields
- verification of the invariant
- insertion into a data structure
- update + invariant preservation

permissions track the state of objects

Permissions

permission = type-level information about a region

permissions evolve during execution:
statements consume and produce permissions

permissions cannot be duplicated

Allocation and Initialization

operation **let region** r: $\mathcal{C}$

- produces $r^{\emptyset}$

operation **let** $x = \textbf{new}\ \mathcal{C}\ [\rho]$

- consumes $\rho^{\emptyset}$
- produces $\rho^{\circ}\{f_1, \cdots, f_k\}$ and owned permissions

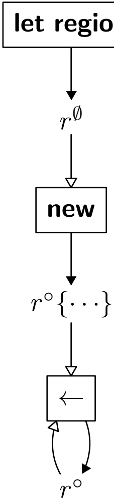
operation $x.f \leftarrow e$ when x: $[\rho]$

- consumes $\rho^{\circ}\{\overline{g}\}$
- produces $\rho^{\circ}\{\overline{g}-f\}$

Allocation: Example

$$\begin{array}{ll}
\textbf{let region } r\colon \mathit{Pair}; & r^{\emptyset} \\
\textbf{let } p = \textbf{new } \mathit{Pair}\ [r]; & r^{\circ}\{\mathit{fst}, \mathit{snd}\} \\
p.\mathit{fst} \leftarrow 42; & r^{\circ}\{\mathit{snd}\} \\
p.\mathit{snd} \leftarrow 69; & r^{\circ}
\end{array}$$

Permission Diagram (so far)



let region

$r^{\emptyset}$

new

$r^{\circ}\{\cdots\}$

$\leftarrow$

r°

Packing and Unpacking

if y: $[\rho]$

operation **pack** y

- consumes ρ° and owned permissions
- produces $\rho^\times$
- requires the invariant of y as a pre-condition

operation **unpack** y

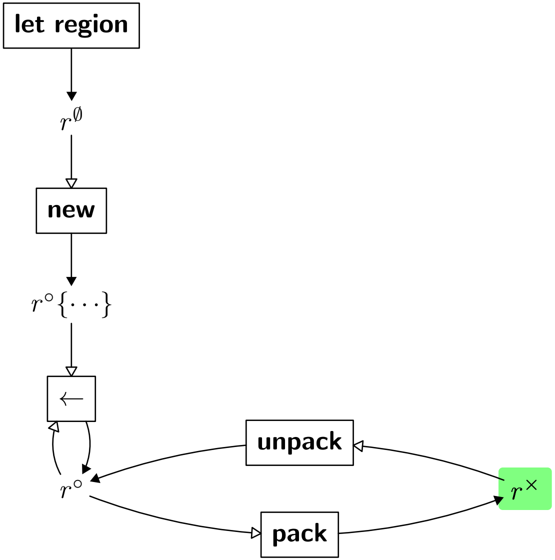
- consumes $\rho^\times$
- produces ρ° and owned permissions

note: ρ° required to modify $y.f$
$\Longrightarrow$ if $\rho^\times$ available, then the invariant of y holds

Example: Incrementing a Pair

```
fun incrPair [r: Pair] (p: [r]): unit
  consumes $r^\times$
  produces $r^\times$
{
  unpack p;                                               $r^\circ$
  p.fst $\leftarrow$ p.fst + 1;                                    $r^\circ$
  p.snd $\leftarrow$ p.snd + 1;                                    $r^\circ$
  pack p; (* invariant must hold *)                       $r^\times$
}
```

Permission Diagram (so far)



Adoption: From Singleton to Group

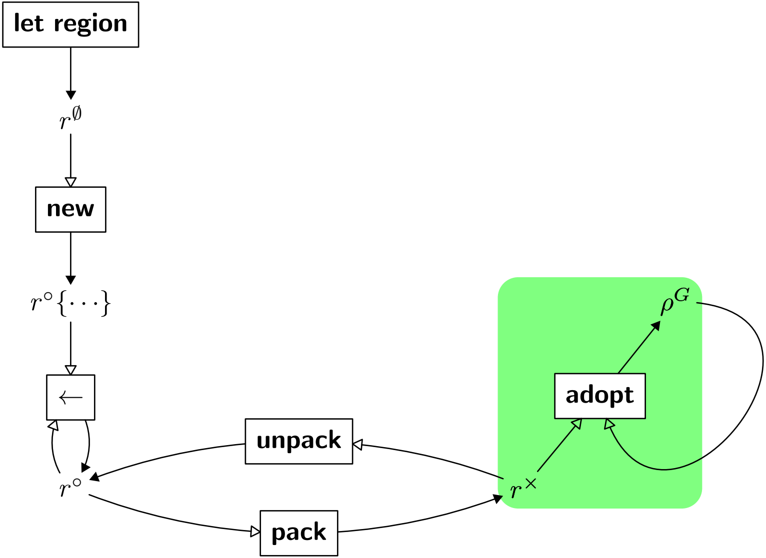
if x: $[\sigma]$

operation **adopt** x: σ **as** ρ

- consumes $\sigma^{\times}$ and ρ^{G}
- produces ρ^{G}
- type of x becomes $[\rho]$

x is then both in σ and ρ
region σ is disabled

Permission Diagram



Focus: From Group to Singleton

operation **focus** x: ρ **as** σ when x: $[\rho]$

- consumes ρ^G and $\sigma^\emptyset$
- produces $\sigma \multimap \rho$ and $\sigma^\times$
- type of x becomes $[\sigma]$

x is then both in σ and ρ
region ρ is temporarily disabled

operation **unfocus** x: σ **as** ρ when x: $[\sigma]$

- consumes $\sigma \multimap \rho$ and $\sigma^\times$
- produces ρ^G
- type of x becomes $[\rho]$

region ρ is re-enabled
region σ is disabled

Aliased or Not Aliased?

p and q may be aliased:

fun f $[r\colon Pair]$ $(p\colon [r],\ q\colon [r])\colon unit$
consumes r^G
produces r^G

p and q cannot be aliased:

fun f $[r_p\colon Pair,\ r_q\colon Pair]$ $(p\colon [r_p],\ q\colon [r_q])\colon unit$
consumes ${r_p}^G\ {r_q}^G$
produces ${r_p}^G\ {r_q}^G$

Ownership

locations may own regions

```
class LongPairOwn
{
  single $r_1$: Long;
  single $r_2$: Long;
  fst: [$r_1$];
  snd: [$r_2$];
  invariant $fst.value < snd.value$;
}
```

invariant can only mention owned objects (enforced by typing)

Allocation With Ownership

$\textbf{let region}\ r\text{: } \mathit{LongPairOwn};$	$r^{\emptyset}$
$\textbf{let}\ p = \textbf{new}\ \mathit{LongPairOwn}\ [r];$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\emptyset}\ p.r_2{}^{\emptyset}$
$\textbf{let}\ \mathit{fst} = \textbf{new}\ \mathit{Long}\ [p.r_1];$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\circ}\{\mathit{value}\}\ p.r_2{}^{\emptyset}$
$\mathit{fst}.\mathit{value} \leftarrow 42;$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\circ}\ p.r_2{}^{\emptyset}$
$\textbf{pack}\ \mathit{fst};$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\times}\ p.r_2{}^{\emptyset}$
$\textbf{let}\ \mathit{snd} = \textbf{new}\ \mathit{Long}\ [p.r_2];$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\times}\ p.r_2{}^{\circ}\{\mathit{value}\}$
$\mathit{snd}.\mathit{value} \leftarrow 69;$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\times}\ p.r_2{}^{\circ}$
$\textbf{pack}\ \mathit{snd};$	$r^{\circ}\{\mathit{fst}, \mathit{snd}\}\ p.r_1{}^{\times}\ p.r_2{}^{\times}$
$p.\mathit{fst} \leftarrow \mathit{fst};$	$r^{\circ}\{\mathit{snd}\}\ p.r_1{}^{\times}\ p.r_2{}^{\times}$
$p.\mathit{snd} \leftarrow \mathit{snd};$	$r^{\circ}\ p.r_1{}^{\times}\ p.r_2{}^{\times}$
$\textbf{pack}\ p;$	$r^{\times}$

Ownership: Summary

allows invariants to depend on owned fields

- need to unpack p to modify $p.fst.value$

structures the heap using an ownership tree

Heap Coherence

we define a memory model and semantics for Capucine

we define coherence of a heap w.r.t. available permissions

- empty regions are empty
- singleton regions have exactly one location
- locations in closed regions verify their invariant
- ...

Coherence Preservation

Theorem (Coherence Preservation)
Coherence of the heap is preserved through execution of a well-typed program.

Summary and Contributions

take the existing notion of regions and permissions

- control aliasing

my contributions

- use permissions to control invariants
- add ownership
- add region parameters to classes
- add region polymorphism
- use inference to guess some operations
  - pack, unpack, adoption, focus, unfocus

Contents

Introduction

The Capucine Language

Computing Verification Conditions
- Use Regions to Separate Pointers
- Prefix Trees
- Experiments
- Progress
- Conclusion

Conclusion

The Why Intermediate Language

the Why Language

- ML-like programs (without higher order)
- first-order logic
- references, with no aliasing
- computes weakest-precondition

encode Capucine programs as Why programs

- challenge: encode memory model to support aliasing

Computing Verification Conditions

encode locations using an abstract type

type *location*

encode each region using a map

type *heap* (α)
logic *select* (*heap* (α), *location*): α
logic *store* (*heap* (α), *location*, α): *heap* (α)

encode objets as records

- each field encoded as a field
- each owned region encoded as a field of type *heap*

Example: Two Regions (Capucine)

```
class Long = { value: int }

fun incr2 [$r_1$: Long, $r_2$: Long] ($i$: [$r_1$], $j$: [$r_2$])
  consumes $r_1^\circ$ $r_2^\circ$
  produces $r_1^\circ$ $r_2^\circ$
  post $i.value = \mathbf{old}(i.value) + 1$
{
  $i.value \leftarrow i.value + 1$;
  $j.value \leftarrow j.value + 1$;
}
```

Example: Two Regions (Why)

type *Long* = { *value*: *int* }

let *incr2* (r_1: **ref** (*heap* (*Long*)), r_2: **ref** (*heap* (*Long*)),
i: *location*, *j*: *location*)
{ *true* }
r_1 := *store* (!r_1, *i*, { *value* = *select* (!r_1, *i*).*value* + 1 });
r_2 := *store* (!r_2, *j*, { *value* = *select* (!r_2, *j*).*value* + 1 });
{ *select* (!r_1, *i*).*value* = *select* (**old**(!r_1), *i*).*value* + 1 }

Issue

current translation: pros

- modify region $\Longrightarrow$ other regions untouched

current translation: cons

- modify owned region $\Longrightarrow$ modify root region

Flatten Ownership Tree

Burstall-Bornat component-as-array model

- one heap per field

idea: extend it to ownership trees

Flatten Ownership Tree

```
type Long = { value: int }
type LongPairOwn = {
  r1: heap (Long);
  r2: heap (Long);
  fst: location;
  snd: location
}
r: ref (heap (LongPairOwn))
```

becomes

```
r_r1_value: ref (heap (heap (int)))
r_r2_value: ref (heap (heap (int)))
r_fst: ref (heap (location))
r_snd: ref (heap (location))
```

Simplify Singleton Regions

r1 and r2 are singleton

$$\boxed{\begin{array}{l} r\_r1\_value\text{: }\mathbf{ref}\ (heap\ (heap\ (int))) \\ r\_r2\_value\text{: }\mathbf{ref}\ (heap\ (heap\ (int))) \end{array}}$$

becomes

$$\boxed{\begin{array}{l} r\_r1\_value\text{: }\mathbf{ref}\ (heap\ (int)) \\ r\_r2\_value\text{: }\mathbf{ref}\ (heap\ (int)) \end{array}}$$

Flatten Ownership Tree

```
p.fst.value ← 42
```

without flattening:

```
r := store (!r, p,
  { select (!r, p) with
    r1 = store (select (!r, p).r1, select (!r, p).fst,
      { select (select (!r, p).r1, select (!r, p).fst)
        with value = 42 }) })
```

with flattening and singleton simplification:

```
r_r1_value := store(!r_r1_value, p, 42)
```

Flattening: Issue

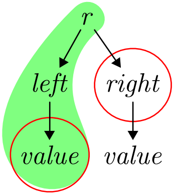
big data structures
$\Longrightarrow$ huge number of leaves in ownership tree
$\Longrightarrow$ huge number of references

recursive data structures
$\Longrightarrow$ infinite number of references

Prefix Tree

idea: only flatten what is used locally

```
fun incrLeft
  [r: LongPairOwn] (p: [r]):
  unit
{
  let x = p.left;
  x.value ← 42;
}
```



node *r* is flattened ⟹ references *r_left* and *r_right*

node *r_left* is flattened ⟹ reference *r_left_value*

Experiments

Alt-Ergo (10s timeout)

	without flattening	with flattening
Course	14s + 1 timeout	1.2s + 1 timeout
Sparse Arrays (*)	120s	26s

Z3 (10s timeout)

	without flattening	with flattening
Course	2s + 7 timeouts	1s + 3 timeouts
Sparse Arrays (*)	96s + 10 timeouts	23s + 3 timeouts

* Sparse Arrays = part of VACID-0 challenge [Leino 2010]
(involves invariants and complex data structures)

Progress

Theorem (Progress)
Assume a well-typed Capucine program, whose proof obligations have been proven. The program executes with no error. In particular, it verifies its specification.

Summary and Contributions

previous work: use regions to separate pointers

- one map per group region
- one (location, value) pair per singleton region

my contributions

- apply this method with:
  - allocation
  - polymorphism
  - ownership
- use prefix trees to achieve more separation
  - experiments show this greatly helps automatic provers

Contents

Expressiveness vs. Automation

where does Capucine stand?

- region annotations in function prototypes
- no proof obligations for invariants except when packing
- inference of some pack, unpack, adopt, focus, unfocus
- type information can be used in hypotheses
  (invariants, region of pointers, freshness)

Future Work

from mainstream languages to Capucine

- annotation language?
- translation of data structures (Java classes, C unions, mutable records...)?

inference mechanism

- global analysis?

combine with other approaches

- separation logic to describe group region contents?