Computer Arithmetic
Aliasing issues: Call by reference, Pointer programs

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

24 janvier 2014

Outline

Exercises from last lecture

Computer Arithmetic
- Handling Machine Integers
- Floating-Point Computations

Aliasing Issues
- Call by Reference
- Pointer Programs

Example: McCarthy's 91 Function

$$f91(n) = \texttt{if } n \leq 100 \texttt{ then } f91(f91(n+11)) \texttt{ else } n - 10$$

Exercise: find adequate specifications.

```
let fun f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n ≤ 100 then f91(f91(n + 11)) else n - 10
```

See `f91.mlw`

Exercises

- Implement and prove binary search using a immediate exit:

  $low = 0$; $high = n - 1$;
  while $low \leq high$:
  let m be the middle of low and $high$
  if $a[m] = v$ then return m
  if $a[m] < v$ then continue search between m and $high$
  if $a[m] > v$ then continue search between low and m

  See `bin_search.mlw`

Exercise: Selection Sort

```
val a: array real

let fun sort ()
  writes a
  ensures ?
body ?
```

1. Formalize postconditions:
   - array in increasing order between 0 and $n - 1$,
   - array at exit is a permutation of the array at entrance.
2. Implement and prove selection sort algorithm:

   for each i from 0 to $n - 1$:
   find index idx of the min element between i and $n - 1$
   swap elements at indexes i and idx

See `sel_sort.mlw`

Outline

Exercises from last lecture

Computer Arithmetic
- Handling Machine Integers
- Floating-Point Computations

Aliasing Issues
- Call by Reference
- Pointer Programs

Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may *overflow*
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$

Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may *overflow*
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$
- floating-point numbers (32-, 64-bit):
  - *overflows*
    - $2 \times 2 \times \cdots \times 2 \rightarrow +inf$
    - $-1/0 \rightarrow -inf$
    - $0/0 \rightarrow \texttt{NaN}$

Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may *overflow*
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$
- floating-point numbers (32-, 64-bit):
  - *overflows*
    - $2 \times 2 \times \cdots \times 2 \rightarrow +inf$
    - $-1/0 \rightarrow -inf$
    - $0/0 \rightarrow$ `NaN`
  - *rounding errors*
    - $\underbrace{0.1 + 0.1 + \cdots + 0.1}_{10 times} = 1.0 \rightarrow$ `false`

      (because $0.1 \rightarrow 0.100000001490116119384765625$ in 32-bit)

See also `arith.c`

Some Numerical Failures

(see more at
http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.

Some Numerical Failures

(see more at
http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a *rounding error* is discovered.

Some Numerical Failures

(see more at
http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a *rounding error* is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an *overflow*: insurance cost is $500M.

Some Numerical Failures

(see more at
http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
- 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a *rounding error* is discovered.
- 1995, Ariane 5 explodes during its maiden flight due to an *overflow*: insurance cost is $500M.
- 2007, Excel *displays* 77.1×850 as 100000.

Some Numerical Failures

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.

  Internal clock ticks every *0.1 second*.
  Time is tracked by *fixed-point arith.*: $0.1 \simeq 209715 \cdot 2^{-24}$.
  Cumulated skew after 24h: -0.08s, distance: 160m.
  System was supposed to be rebooted periodically.

- 2007, Excel displays 77.1×850 as 100000.

  Bug in *binary/decimal conversion*.
  Failing inputs: 12 FP numbers.
  Probability to uncover them by random testing: 10^{-18}.

Integer overflow: example of Binary Search

- Google “Read All About It: Nearly All Binary Searches and Mergesorts are Broken”

```
let l = ref 0 in
let u = ref (length a) - 1 in
while l ≤ u do
  let m = (l + u) / 2 in
  ...
```

$l + u$ may overflow with large arrays!

Goal

prove that a program is safe with respect to overflows

Target Type: int32

- 32-bit signed integers in two-complement representation: integers between -2^{31} and $2^{31} - 1$.
- If the mathematical result of an operation fits in that range, that is the computed result.
- Otherwise, an overflow occurs.
  Behavior depends on language and environment:
  modulo arith, saturated arith, abrupt termination, etc.

A program is safe if no overflow occurs.

Safety Checking

Idea: replace all arithmetic operations by abstract functions with preconditions. $x + y$ becomes int32_add(x, y).

```
val int32_add(x: int, y: int): int
  requires -2^31 ≤ x + y < 2^31
  ensures result = x + y
```

Unsatisfactory: range contraints of integer must be added explicitly everywhere

Safety Checking, Second Attempt

Idea: replace

- type *int* with an abstract type *int32* coercible to it,
- all operations by abstract functions with preconditions,

and add an axiom about the range of *int32*.

```
predicate in_int32 (n:int) =  -2^31 ≤ n < 2^31

type int32
function of_int32(x: int32): int
axiom bounded_int32: forall x: int32. in_int32(of_int32(x))

val int32_add(x: int32, y: int32): int32
  requires in_int32(of_int32(x) + of_int32(y))
  ensures of_int32(result) = of_int32(x) + of_int32(y)
```

Binary Search with overflow checking

See `bin_search_int32.mlw`

Binary Search with overflow checking

See `bin_search_int32.mlw`

Application

Used for translating mainstream programming language into Why3:

- From C to Why3: Frama-c, Jessie plug-in
  See `bin_search.c`
- From Java to Why3: Krakatoa
- From Ada to Why3: Spark2014

Floating-Point Arithmetic

- Limited range $\Rightarrow$ exceptional behaviors.
- Limited precision $\Rightarrow$ inaccurate results.

Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.
Width: $1 + w_e + w_m = 32$, or 64, or 128.
Bias: $2^{w_e - 1} - 1$. Precision: $p = w_m + 1$.

A floating-point datum

sign s	biased exponent e' (w_e bits)	mantissa m (w_m bits)

represents

Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.
Width: $1 + w_e + w_m = 32$, or 64, or 128.
Bias: $2^{w_e - 1} - 1$. Precision: $p = w_m + 1$.

A floating-point datum

sign s	biased exponent e' (w_e bits)	mantissa m (w_m bits)

represents

- if $0 < e' < 2^{w_e} - 1$, the real $(-1)^s \cdot \overline{1.m'} \cdot 2^{e' - bias}$, normal
- if $e' = 0$,
  - ± 0 if $m' = 0$, zeros
  - the real $(-1)^s \cdot \overline{0.m'} \cdot 2^{-bias+1}$ otherwise, subnormal
- if $e' = 2^{w_e} - 1$,
  - $(-1)^s \cdot \infty$ if $m' = 0$, infinity
  - *Not-a-Number* otherwise. NaN

Floating-Point Data

$$\begin{array}{ccccc} \boxed{1} & & \boxed{11000110} & & \boxed{10010011110000111000000} \\ s & & e & & f \\ \downarrow & & \downarrow & & \downarrow \\ (-1)^s & \times & 2^{e-B} & \times & 1.f \end{array}$$

$$(-1)^1 \times 2^{198-127} \times 1.10010011110000111000000_2$$

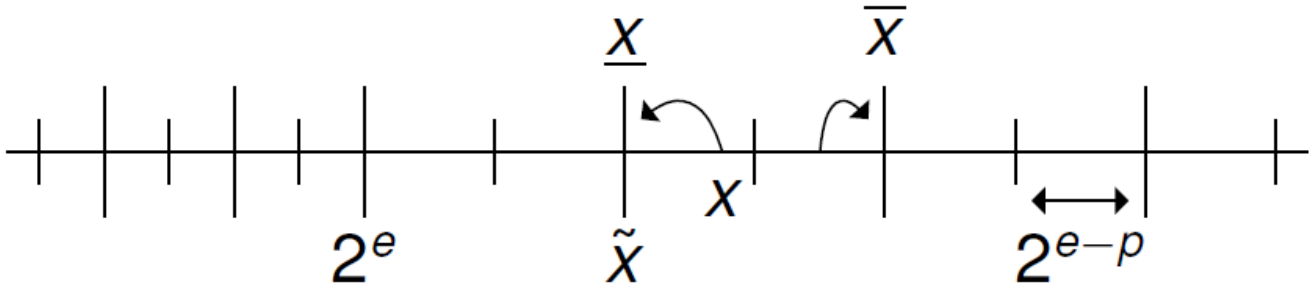
$$-2^{54} \times 206727 \approx -3.7 \times 10^{21}$$

Semantics for the Finite Case

IEEE-754 standard

A floating-point operator shall behave as if it was first computing the infinitely-precise value and then rounding it so that it fits in the destination floating-point format.

Rounding of a real number x:



Overflows are not considered when defining rounding: exponents are supposed to have no upper bound!

Specifications, main ideas

Same as with integers, we specify FP operations
so that no overflow occurs.

```
constant max : real = 0x1.FFFFFEp127
predicate in_float32 (x:real) = abs x ≤ max
type float32
function of_float32(x: float32): real
axiom float32_range: forall x: float32. in_float32 (of_float32 x)

function round32(x: real): real
(* ... axioms about round32 ... *)

val float32_add(x: float32, y: float32): float32
  requires in_float32(round32(of_float32 x - of_float32 y))
  ensures of_float32 result = round32 (of_float32 x - of_float32 y)
```

Specifications in practice

- Several possible rounding modes
- many axioms for `round32`, but incomplete anyway
- Specialized prover: Gappa http://gappa.gforge.inria.fr/

Demo: `clock_drift.c`

Outline

Exercises from last lecture

Computer Arithmetic
- Handling Machine Integers
- Floating-Point Computations

Aliasing Issues
- Call by Reference
- Pointer Programs

Reminder of former lectures

Compound data structures can be *modeled* using expressive specification languages

- Defined functions and predicates
- Product types (records)
- Sum types (lists, trees)
- Axiomatizations (arrays, sets)

Important points:

- *pure* types, no internal “in-place” assignment
- Mutable variables = *references to pure types*

Aliasing

Aliasing = two different “names” for the same mutable data

Two sub-topics:

- Call by reference
- Pointer programs

Need for call by reference

Example: stacks of integers

```
type stack = list int

val s:ref stack

let fun push(x:int):unit
  writes s
  ensures s = Cons(x,s@Old)
  body ...

let fun pop(): int
  requires s ≠ Nil
  writes s
  ensures result = head(s@Old) ∧ s = tail(s@Old)
```

Need for call by reference

If we need two stacks in the same program:

- We don't want to write the functions twice!

We want to write

```
type stack = list int

let fun push(s:ref stack,x:int): unit
  writes s
  ensures s = Cons(x,s@Old)
  ...

let fun pop(s:ref stack):int)
  ...
```

Call by Reference: example

```
val s1,s2: ref stack

let fun test():
  writes s1, s2
  ensures result = 13 ∧ head(s2) = 42
  body push(s1,13); push(s2,42); pop(s1)
```

- See file `stack1.mlw`

Aliasing problems

```
let fun test(s3,s4: ref stack) : unit
  writes s3, s4
  ensures { head(s3) = 42 ∧ head(s4) = 13 }
  body push(s3,42); push(s4,13)

let fun wrong(s5: ref stack) : int
  writes s5
  ensures { head(s5) = 42 ∧ head(s5) = 13 }
              something's wrong !?
  body test(s5,s5)
```

Aliasing is a major issue

Deductive Verification Methods like Hoare logic, Weakest Precondition Calculus implicitly require absence of aliasing

Syntax

- Declaration of functions: (references first for simplicity)

  let fun $f(y_1 : \text{ref } \tau_1, \ldots, y_k : \text{ref } \tau_k, x_1 : \tau'_1, \ldots, x_n : \tau'_n)$:
  $\ldots$

- Call:

  $f(z_1, \ldots, z_k, e_1, \ldots, e_n)$

  where each z_i must be a reference

Operational Semantics

Intuitive semantics, by substitution:

$$\frac{\Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre} \quad \mathit{Body}' = \mathit{Body}[y_j \leftarrow z_j]}{\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (\mathit{Old} : \texttt{frame}(\Pi', \mathit{Body}', \mathit{Post}))}$$

- The body is executed, where each occurrence of reference parameters are replaced by the corresponding reference argument.
- Not a “practical” semantics, but that’s not important. . .

Operational Semantics

Variant: Semantics by copy/restore:

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models \mathit{Pre}}{\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \rightsquigarrow \Sigma', \Pi, (\mathit{Old} : \texttt{frame}(\Pi', \mathit{Body}, \mathit{Post}))}$$

$$\frac{\Sigma, \Pi' \models P[\text{result} \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]}{\Sigma, \Pi, (\texttt{frame}(\Pi', v, P)) \rightsquigarrow \Sigma', \Pi, v}$$

Operational Semantics

Variant: Semantics by copy/restore:

$$\frac{\Sigma' = \Sigma[y_j \leftarrow \Sigma(z_j)] \quad \Pi' = \{x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}\} \quad \Sigma, \Pi' \models Pre}{\Sigma, \Pi, f(z_1, \ldots, z_k, t_1, \ldots, t_n) \rightsquigarrow \Sigma', \Pi, (Old : \texttt{frame}(\Pi', Body, Post))}$$

$$\frac{\Sigma, \Pi' \models P[\text{result} \leftarrow v] \quad \Sigma' = \Sigma[z_j \leftarrow \Sigma(y_j)]}{\Sigma, \Pi, (\texttt{frame}(\Pi', v, P)) \rightsquigarrow \Sigma', \Pi, v}$$

Warning: not the same semantics !

Difference in the semantics

```
val g : ref int

let fun f(x:ref int):unit
  body x := 1; x := g+1

let fun test():unit
  body g:=0; f(g)
```

After executing test:

- Semantics by substitution: `g = 2`
- Semantics by copy/restore: `g = 1`

Aliasing Issues (1)

```
let fun f(x:ref int, y:ref int):
  writes x, y
  ensures x = 1 ∧ y = 2
  body x := 1; y := 2

val g : ref int

let fun test():
  body
    f(g,g);
    assert g = 1 ∧ g = 2 (* ???? *)
```

- Aliasing of reference parameters

Aliasing Issues (2)

```
val g1 : ref int
val g2 : ref int

let fun p(x:ref int):
  writes g1, x
  ensures g1 = 1 ∧ x = 2
  body g1 := 1; x := 2

let fun test():
  body
    p(g2); assert g1 = 1 ∧ g2 = 2; (* OK *)
    p(g1); assert g1 = 1 ∧ g1 = 2; (* ??? *)
```

- Aliasing of a global variable and reference parameter

Aliasing Issues (3)

```
val g : ref int

val fun f(x:ref int):unit
  writes x
  ensures x = g + 1
  (* body x := 1; x := g+1 *)

let fun test():unit
  ensures { g = 1 or 2 ? }
  body g := 0; f(g)
```

- Aliasing of a read reference and a written reference

Aliasing Issues (3)

New need in specifications

Need to *specify read references in contracts*

```
val g : ref int

val f(x:ref int):unit
  reads g          (* new clause in contract *)
  writes x
  ensures x = g + 1
  (* body x := 1; x := g+1 *)

let fun test():unit
  ensures { g = ? }
  body g := 0; f(g)
```

- See file `stack2.mlw`

Typing: Alias-Freedom Conditions

For a function of the form

$f(y_1 : \tau_1, ..., y_k : \tau_k, ...) : \tau$:
  writes $\vec{w}$
  reads $\vec{r}$

Typing rule for a call to f:

$$\frac{\ldots \quad \forall ij, i \neq j \rightarrow z_i \neq z_j \quad \forall i, j, z_i \neq w_j \quad \forall i, j, z_i \neq r_j}{\ldots \vdash f(z_1, \ldots, z_k, ...) : \tau}$$

- effective arguments z_j must be distinct
- effective arguments z_j must not be read nor written by f

Proof Rules

Thanks to restricted typing:

- Semantics by substitution and by copy/restore coincide
- Hoare rules remain correct
- WP rules remain correct

New references

- Need to return newly created references
- Example: stack continued

```
let fun create():ref stack
  ensures result = Nil
  body (ref Nil)
```

- Typing should require that a returned reference is always *fresh*

See file `stack3.mlw`

Pointer programs

- We drop the hypothesis “no reference to reference”
- Allows to program on *linked data structures*. Example (in the C language):

```
struct List { int data; list next; } *list;
while (p <> NULL) { p->data++; p = p->next }
```

- “In-place” assignment
- References are now *values* of the language: “pointers” or “memory addresses”

We need to handle aliasing problems differently

Syntax

- For simplicity, we assume a language with pointers to records
- Access to record field: `e→f`
- Update of a record field: `e→f := e'`

Operational Semantics

- New kind of values: *loc* = the type of pointers
- A special value *null* of type loc is given
- A program state is now a pair of
  - a *store* which maps variables identifiers to values
  - a *heap* which maps pairs (loc, field name) to values
- Memory access and updates should be proved safe (no "null pointer dereferencing")
- For the moment we forbid allocation/deallocation

Component-as-array trick

If:

- a program is *well-typed*
- The set of *all field names are known*

then the heap can be also seen as *a finite collection of maps*, one for each field name

- map for a field of type τ maps loc to values of type τ

This “trick” allows to *encode pointer programs* into Why3 programs

- Use maps indexed by locs instead of integers

Component-as-array model

```
type loc
constant null : loc

val acc(field: ref (map loc α),l:loc) : α
  requires l ≠ null
  reads field
  ensures result = select(field,l)

val upd(field: ref (map loc α),l:loc,v:α):unit
  requires l ≠ null
  writes field
  ensures field = store(field@Old,l,v)
```

Encoding:

- Access to record field: `e→f` becomes `acc(f,e)`
- Update of a record field:
  `e→f := e'` becomes `upd(f,e,e')`

Example

- In C

```
struct List { int data; list next; } *list;

while (p <> NULL) { p->data++; p = p->next }
```

- In Why3

```
val data: ref (map loc int)
val next: ref (map loc loc)

while p ≠ null do
   upd(data,p,acc(data,p)+1);
   p := acc(next,p)
```

In-place List Reversal

A la C/Java:

```
list reverse(list l) {
   list p = l;
   list r = null;
   while (p != null) {
     list n = p->next;
     p->next = r;
     r = p;
     p = n
  }
  return r;
}
```

In-place Reversal in our Model

```
let fun reverse (l:loc) : loc =
   let p = ref l in
   let r = ref null in
   while (p ≠ null) do
     let n = acc(next,p) in
     store(next,p,r);
     r := p;
     p := n
  done;
  r
```

Goals:

- Specify the expected behavior of `reverse`
- Prove the implementation

Specifying the function

Predicate `list_seg`$(p, next, p_M, q)$: p points to a list of nodes p_M that ends at q.

$$p = p_0 \overset{next}{\mapsto} p_1 \cdots \overset{next}{\mapsto} p_k \overset{next}{\mapsto} \mathrm{q}$$

$$p_M = Cons(p_0, Cons(p_1, \cdots Cons(p_k, Nil) \cdots))$$

p_M is the *model list* of p

```
inductive list_seg(loc,map loc loc,list loc,loc) =
| list_seg_nil:
  forall p:loc, next:map loc loc. list_seg(p,next,Nil,p)
| list_seg_cons:
  forall p q:loc, next:map loc loc, pM:list loc.
    p ≠ null ∧ list_seg(select(next,p),next,pM,q) →
      list_seg(p,next,Cons(p,pM),q)
```

Specification

- pre: input l well-formed:

$$\exists l_M.\texttt{list\_seg}(l, next, l_M, null)$$

- post: output well-formed:

$$\exists r_M.\texttt{list\_seg}(result, next, r_M, null)$$

  and

$$r_M = rev(l_M)$$

Issue: quantification on l_M is global to the function

- Use *ghost* variables

Annotated In-place Reversal

```
let fun reverse (l:loc) (ghost lM:list loc) : loc =
  requires list_seg(l,next,lM,null)
  writes next
  ensures list_seg(result,next,rev(lM),null)
  body
   let p = ref l in
   let r = ref null in
   while (p ≠ null) do
     let n = acc(next,p) in
     store(next,p,r);
     r := p;
     p := n
  done;
  r
```

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

Local ghost variables p_M, r_M

$$\texttt{list\_seg}(p, next, p_M, null)$$

$$\texttt{list\_seg}(r, next, r_M, null)$$

In-place Reversal: loop invariant

```
while (p ≠ null) do
  let n = acc(next,p) in
  store(next,p,r);
  r := p;
  p := n
```

Local ghost variables p_M, r_M

$$\texttt{list\_seg}(p, next, p_M, null)$$

$$\texttt{list\_seg}(r, next, r_M, null)$$

$$append(rev(p_M), r_M) = rev(l_M)$$

See file `linked_list_rev.mlw`

Needed lemmas

- Need to show that `list_seg` remains true when `next` is updated:

```
lemma list_seg_frame: forall next1 next2:map loc loc,
  p q v: loc, pM:list loc.
  list_seg(p,next1,pM,null) ∧
  next2 = store(next1,q,v) ∧
  ¬ mem(q,pM) → list_seg(p,next2,pM,null)
```

- For that, need to show that p_M, r_M are *disjoint*
- For that, need to show that a rep list do not contain repeted elements

```
lemma list_seg_no_repet:
  forall next:map loc loc, p: loc, pM:list loc.
    list_seg(p,next,pM,null) → no_repet(pM)
```

Exercise

The algorithm that appends two lists *in place* follows this pseudo-code:

```
append(l1,l2 : loc) : loc
  if l1 is empty then return l2;
  let ref p = l1 in
  while p→next is not null do p := p→ next;
  p → next := l2;
  return l1
```

1. Specify a post-condition giving the list models of both `result` and `l2` (add any ghost variable needed)
2. Which pre-conditions and loop invariants are needed to prove this function?

See `linked_list_app.mlw`

Advertising next lectures

- Reasoning on pointer programs using the component-as-array trick is complex
  - need to state and prove *frame* lemmas
  - need to specify many *disjointness* properties
  - even harder is the handling of *memory allocation*
- *Separation Logic* is another approach to reason on heap memory
  - memory resources *explicit* in formulas
  - frame lemmas and disjointness properties are internalized