LOPSTR 2010

Verification of the Schorr-Waite algorithm

From Trees to graphs

Mathieu Giorgino Martin Strecker

Ralph Matthes Marc Pantel

Institut de Recherche en Informatique de Toulouse

24/07/2010

Outline

1. Setting
   - Introduction
   - The Schorr-Waite algorithm
2. Schorr-Waite on trees
   - Definitions
   - Verification
3. Memory model
   - Recall of monads
   - State reader/transformer monads in Isabelle
4. Implementation on trees
   - Definitions
   - Verification
5. Schorr-Waite on trees with pointers
   - Choosing the spanning tree
6. Implementation on graphs
   - Verification
7. Conclusion

Motivation

Our goal

Verification of algorithms manipulating pointers.

- Static analysis of existing code
  - Automated but limited set of verifiable properties
- Specialized proof systems for imperative programs
  - Usually less expressive logics
- Embedding of general purpose proof assistant
  - Code generation (possibly optimized)
  - Greater expressiveness for specification and verification (Higher order)
  - Profits from prover improvements

Previous Work

Conclusions of previous works

- Reason about non-structured graphs make proofs harder
- Automatic verification is only possible on limited structures and properties with a non-elementary complexity.

Hence our approach:

- Using arborescent structures with additional pointers if necessary
- Using a proof assistant allowing a greater expressivity

Why Schorr-Waite ?

An interresting case study

- Numerous formalizations and proofs
- The major part of the graph pointers are modified
- Arbitrary graphs

Only a beginning

- Verification of programs manipulating structured graphs
- Application to Model transformations

Related works on this algorithm

Description

- 1965
- Schorr and Waite [SW67]
- Peter Deutsch

Formalization, Verification

- [Bor00] Bornat (Jape)
- [MN05] Mehta, Nipkow (Isabelle)
- [HM05] Hubert, Marché (Coq)
- [Abr03] Abrial (Atelier B)
- [LRS06] Loginov, Reps, Sagiv (TVLA)

Algorithm features

Purpose

- Marking graphs without using more space (stack, ...)
- Traversing a tree by terminal recursivity and without stack

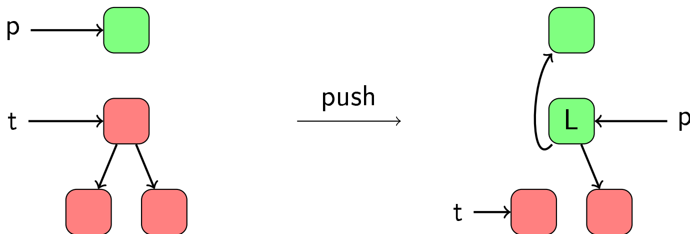
Use

Garbage collector, case study...

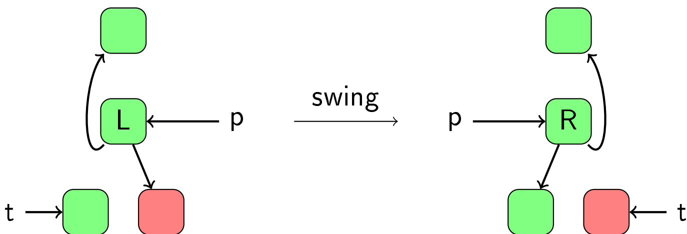
Principle

- Modification of the graph pointers to store the path to the root
- 2 variables containing the pointers :
  - t: to the current node
  - p: to the previously visited node

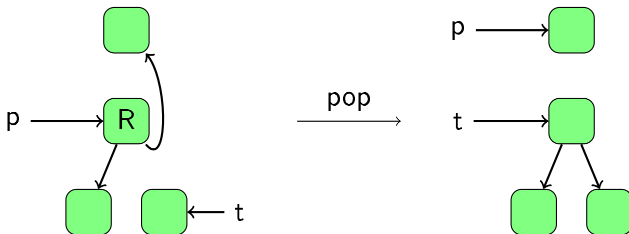
Steps : Push



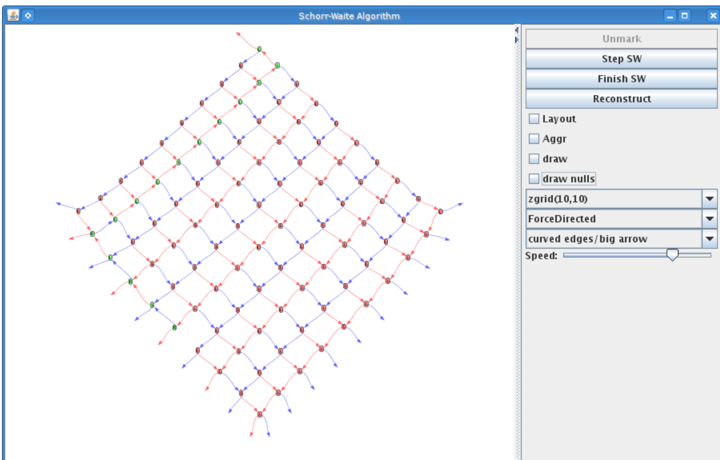
Steps : Swing



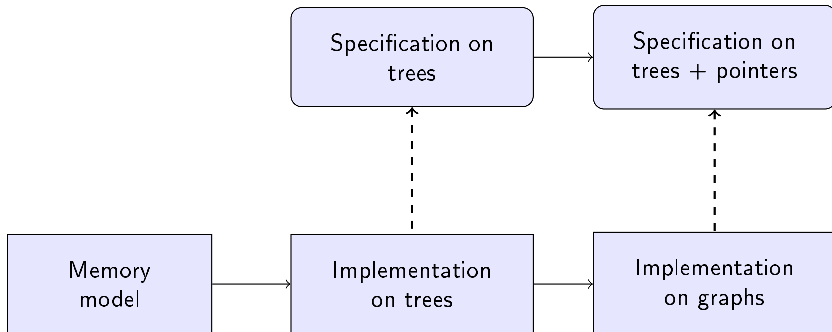
Steps : Pop



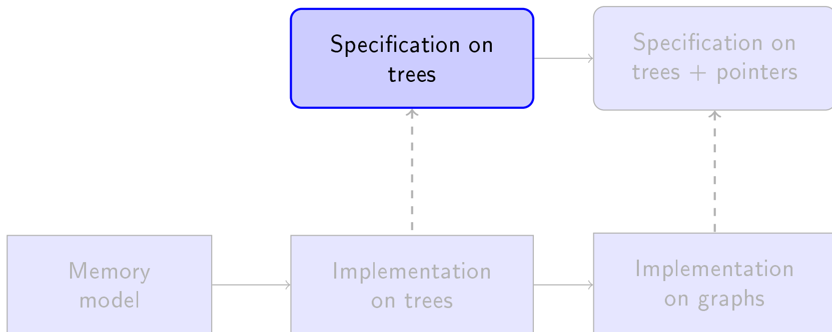
Demo



Outline



Schorr-Waite on trees



Datatypes

We represent trees with the help of 3 datatypes:

trees themselves

datatype ('a,'l) tree =
  Leaf 'l
| Node 'a (('a,'l) tree) (('a,'l) tree)

the 2 directions used to remember the reversed pointer

datatype dir = L | R

the labeled elements of the tree

datatype 'a tag = Tag bool dir 'a

The algorithm

The definition uses 2 auxiliary functions:

the termination condition

```
fun leaf-or-marked where
  leaf-or-marked t = (case t of Leaf - ⇒ True | (Node (Tag m - -) - -) ⇒ m)

fun sw-term :: (('a tag, 'l) tree ∗ ('a tag, 'l) tree) ⇒ bool where
  sw-term (p, t) = (case p of Leaf - ⇒ leaf-or-marked t | - ⇒ False)
```

and its body

```
fun sw-body :: (('a tag, 'l) tree ∗ ('a tag, 'l) tree)
          ⇒ (('a tag, 'l) tree ∗ ('a tag, 'l) tree)
where sw-body (p, t) = (case t of
   (Node (Tag False d v) tlf tr) ⇒ ((Node (Tag True L v) p tr), tlf) (∗ push ∗)
 | - ⇒ (case p of
        Leaf - ⇒ (p, t) (∗ ... ∗)
      | (Node (Tag m L v) pl pr) ⇒ ((Node (Tag m R v) t pl), pr) (∗ swing ∗)
      | (Node (Tag m R v) pl pr) ⇒ (pr, (Node (Tag m R v) pl t)))) (∗ pop ∗)
```

The algorithm - termination

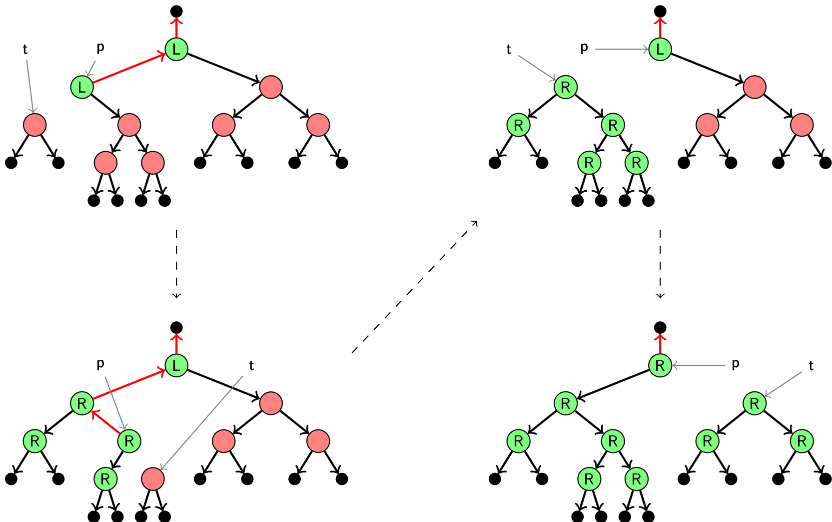
gathered in the complete function

function sw-tr :: (('a tag, 'l) tree * ('a tag, 'l) tree)
⇒ (('a tag, 'l) tree * ('a tag, 'l) tree)
where sw-tr args = (if (sw-term args) then args else sw-tr (sw-body args))

which terminates

termination sw-tr
apply (relation measures [
λ (p,t). unmarked-count p + unmarked-count t,
λ (p,t). left-count p + left-count t,
λ (p,t). size p])
by simp (fastsimp split add: tree.splits tag.splits)

Analysis



Correction

Homogeneity of a tree (property preserved by t)

```
consts t-marked :: bool ⇒ ('a tag, 'l) tree ⇒ bool
primrec
  t-marked m (Leaf rf) = True
  t-marked m (Node n l r) = (case n of (Tag m' d v) ⇒
      ((m ⟶ (d = R)) ∧ m' = m ∧ t-marked m l ∧ t-marked m r))
```

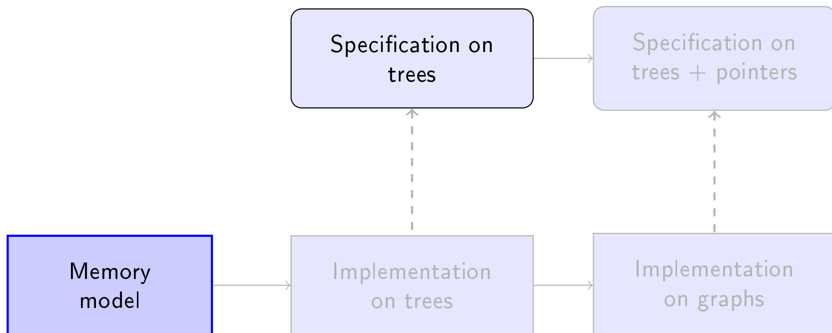
Marking a tree

```
fun mark-all :: bool ⇒ dir ⇒ ('a tag, 'l) tree ⇒ ('a tag, 'l) tree where
  mark-all m d (Leaf rf) = Leaf rf
| mark-all m d (Node (Tag - - v) l r) =
    (Node (Tag m d v) (mark-all m d l) (mark-all m d r))
```

Correction

```
theorem sw-tr-correct:
 t-marked m t ⟹ sw-tr (Leaf rf, t) = (Leaf rf, mark-all True R t)
```

Memory model



History

Monads:

- Element of Category Theory
- Introduced as a computer-sciences tool by Eugenio Moggi [Mog89][Mog91]
- Used by Philip Wadler and Simon Peyton Jones[Wad90][JW93]
- Used intensively in the Haskell language[Pey03]

Definition

A monad is composed of:

- a type constructor: `'a m`
- and two polymorphic operators:
  - return : 'a ⇒ 'a m
  - bind (⊵) : 'a m ⇒ ('a ⇒ 'b m) ⇒ 'b m

following 3 rules:

- left identity: ∀ f v. (return v) ⊵ f = f v
- right identity: ∀ a. (a ⊵ return) = a
- associativity:
  ∀ x f g. (x ⊵ (λ v. f v)) ⊵ g = x ⊵ (λ v. (f v ⊵ g))

Definition

A monad is composed of:

- a type constructor: `'a m`
- and two polymorphic operators:
  - return : 'a $\Rightarrow$ 'a m
  - bind ($\trianglerighteq$) : 'a m $\Rightarrow$ ('a $\Rightarrow$ 'b m) $\Rightarrow$ 'b m

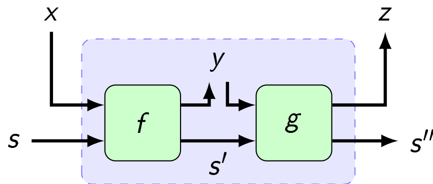
following 3 rules:

- left identity: $\forall$ f v. (return v) $\trianglerighteq$ f $=$ f v
- right identity: $\forall$ a. (a $\trianglerighteq$ return) $=$ a
- associativity:
  $\forall$ x f g. (x $\trianglerighteq$ (λ v. f v)) $\trianglerighteq$ g $=$ x $\trianglerighteq$ (λ v. (f v $\trianglerighteq$ g))

The State-Transformer monad: ('a, 's) ST

avec x $'a$, f $'a \Rightarrow ('b, 's)\ ST$ et g $'b \Rightarrow ('c, 's)\ ST$,

```
doST{
  y ← f x;
  z ← g y;
  returnST z;
}
```



The state s is implicitly transmitted to f and g.

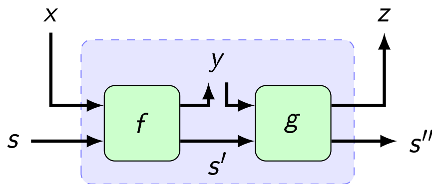
Definition:

- a type constructor:
  - **datatype** ('a, 's) ST = ST ('s ⇒ ('a × 's))
- and 2 operators:
  - returnST a = ST (λ s. (a, s))
  - bindST m f = ST (λ s. (λ (x, s'). runST (f x) s') (runST m s))

The State-Transformer monad: ('a, 's) ST

avec $x^{'a}$, $f^{'a \Rightarrow ('b,\ 's)\ ST}$ et $g^{'b \Rightarrow ('c,\ 's)\ ST}$,

```
doST{
  y ← f x;
  z ← g y;
  returnST z;
}
```



The state s is implicitly transmitted to f and g.

Definition:

- a type constructor:
  - **datatype** ('a, 's) ST = ST ('s ⇒ ('a × 's))
- and 2 operators:
  - returnST a = ST (λ s. (a, s))
  - bindST m f = ST (λ s. (λ (x, s'). runST (f x) s') (runST m s))

('a, 's) SR and ('a, 's) ST

With the state reader monad:

With f $'c \Rightarrow 'b \Rightarrow ('d, 's)\ ST$, a $('a, 's)\ SR$, b $'b$, g $'a \Rightarrow ('b, 's)\ SR$ and h $'d \Rightarrow 'e$:

```
doST {
  x ← f ⟨g ⟨a⟩⟩ b;
  returnST (h x)}
```

⟷

```
ST (λh0. (
  let
    va = runSR a h0;
    vga = runSR (g va) h0;
   (x, h1) = runST (f vga b) h0
  in
    (h x, h1)
))
```

and we add syntax:

```
if (c) {a} else {b}      ⟶   if ⟨doSR{c}⟩ then (doST{a}) else (doST{b})
[v = v0] while (c) {a}   ⟶   whileST (λv. doSR{c}) (λv. doST{a}) v0
while (c) {a}            ⟶   whileST (λ-. doSR{c}) (λ-. doST{a}) ()
```

The memory: ('n, 'v) heap

To manipulate references, we add:

types

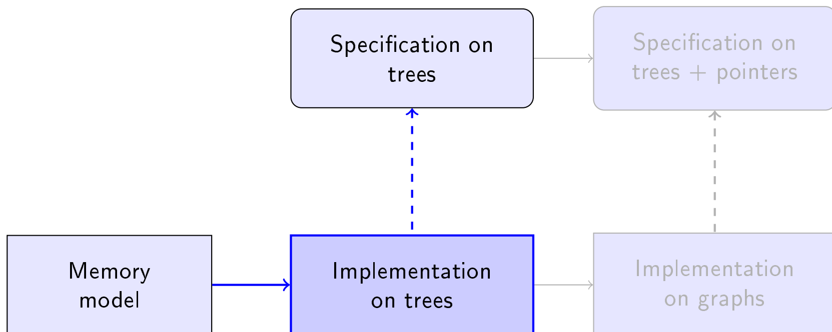
datatype 'n ref = Ref 'n | Null

record ('n, 'v) heap = heap :: 'n ⇒ 'v

and operators

read :: 'n ref ⇒ ('v, ('n, 'v) heap) SR
write (r .= a) :: 'n ref ⇒ v ⇒ (unit, ('n, 'v) heap) ST
get (a·b) :: ('a, 'b) acc ⇒ 'a ⇒ 'b
...

Implementation on trees



Data structures

We define the types:

of the trees/nodes in the memory

datatype ('a, 'r) struct = Struct 'a ('r ref) ('r ref)

of a memory containing trees/nodes

types ('r, 'a) str-heap = ('r, ('a, 'r) struct) heap

and of tree nodes addresses

datatype ('r, 'v) addr = Addr 'r 'v

Implementation

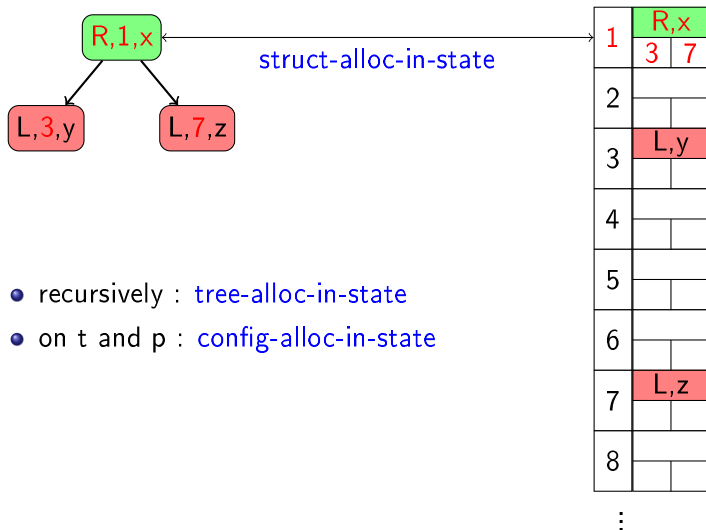
```
constdefs
  sw-impl-body :: ('r ref × 'r ref) ⇒ ('r ref × 'r ref, ('r, 'v tag) str-heap) ST
  sw-impl-body vs == (case vs of (p, t) ⇒ doST {
    if (⟨ null-or-marked t ⟩) {
      (if (⟨ p → ($v oo $dir) ⟩ = L) {    (** swing **)
          let rt = ⟨p → $r⟩;
          p → $r := ⟨p → $l⟩;
          p → $l := t;
          p → ($v oo $dir) := R;
          returnST (p, rt)
      } else {                       (** pop **)
          let rp = ⟨p → $r⟩;
          p → $r := t;
          returnST (rp, p) })
    } else {                         (** push **)
        let rt = ⟨t → $l⟩;
        t → $l := p;
        t → ($v oo $mark) := True;
        t → ($v oo $dir) := L;

        returnST (t, rt) }})
```

Implementation

constdefs
sw-impl-term :: ('r ref × 'r ref) ⇒ (bool, ('r, 'v tag) str-heap) SR
sw-impl-term vs == doSR { (case vs of (ref-p, ref-t) ⇒
(case ref-p of Null ⇒ ⟨ null-or-marked ref-t ⟩ | - ⇒ False))}

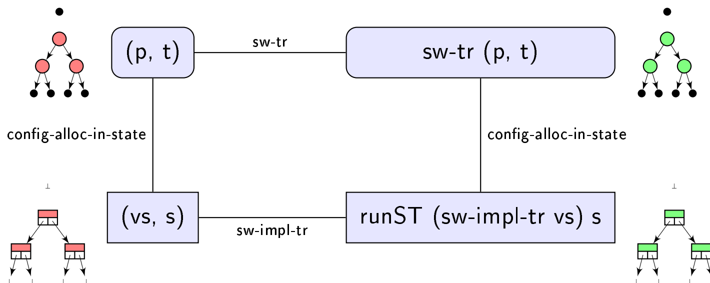
fun sw-impl-tr :: ('r ref × 'r ref) ⇒ ('r ref × 'r ref, ('r, 'v tag) str-heap) ST
where sw-impl-tr pt =
(doST{ [vs = pt] while (¬ ⟨ sw-impl-term vs ⟩) { sw-impl-body vs }})

Allocation



- recursively : tree-alloc-in-state
- on t and p : config-alloc-in-state

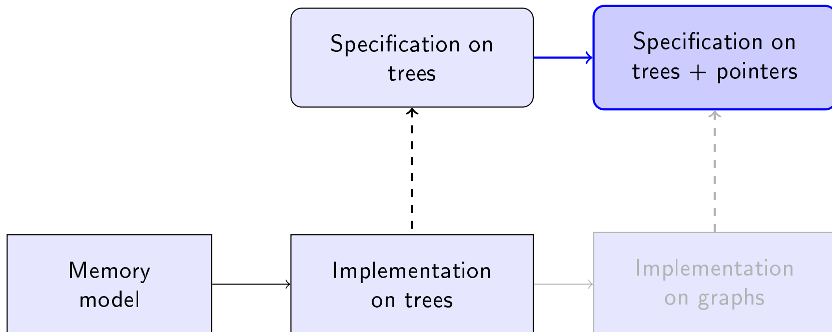
Simulation



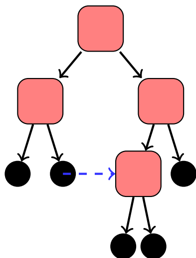
simulation

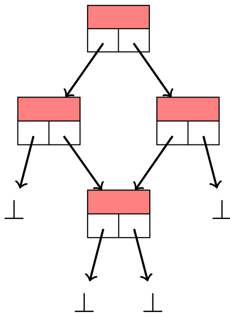
lemma impl-correct:
(∃ m. t-marked m t) ∧ p-marked p ∧ distinct (reach p @ reach t) ∧
config-alloc-in-state addr-of (p, t) (vs,s)
$\Longrightarrow$ config-alloc-in-state addr-of (sw-tr (p, t)) (runST (sw-impl-tr vs) s)

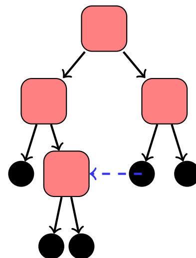
Schorr-Waite on trees with pointers



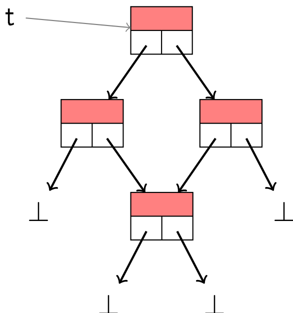
Example: 2 possibility

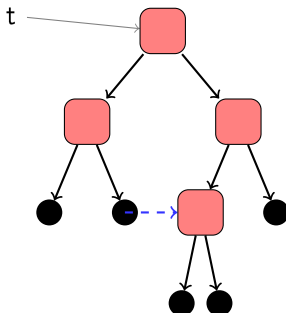




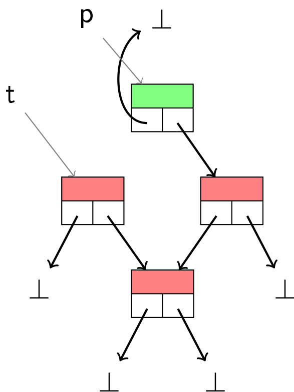


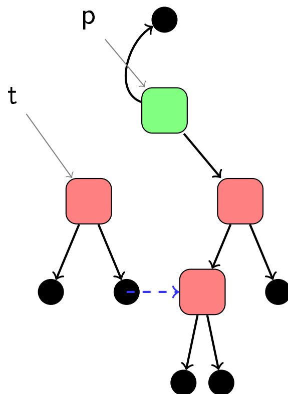
The wrong one!



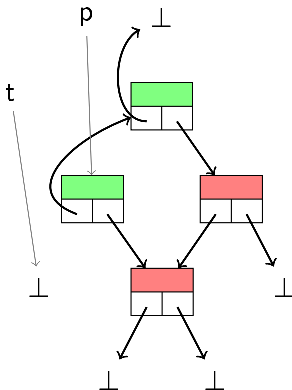


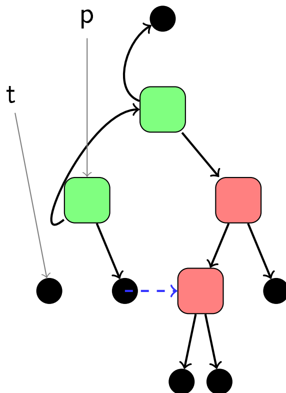
The wrong one!



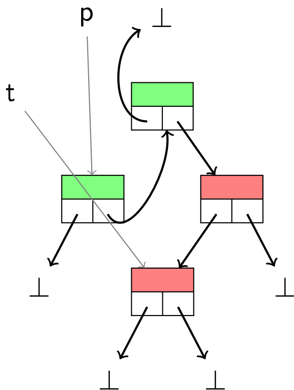


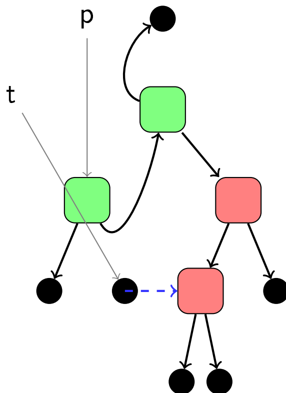
The wrong one!



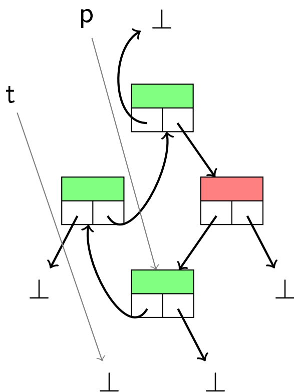


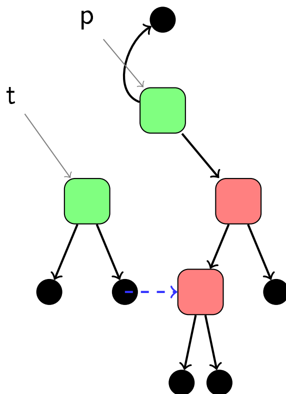
The wrong one!



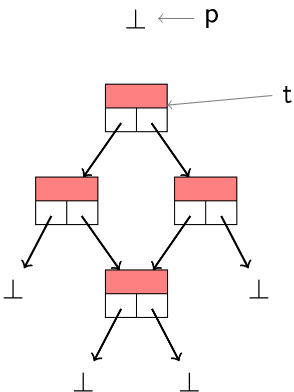


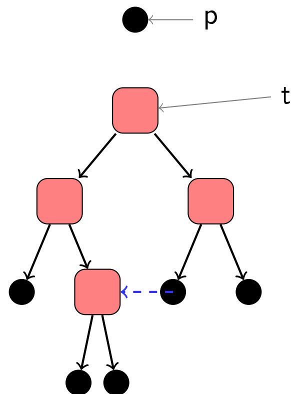
The wrong one!



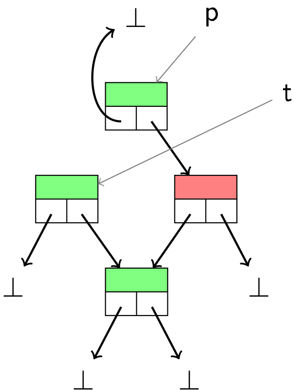


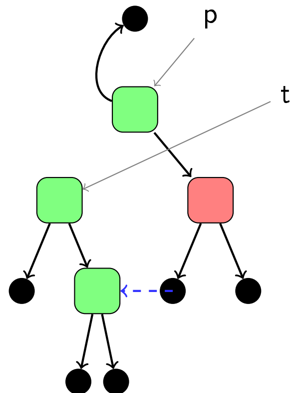
The good one!



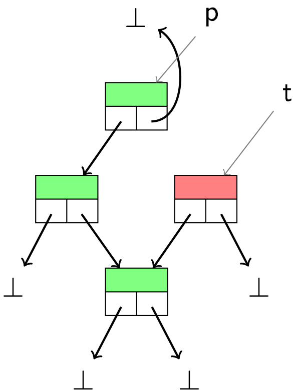


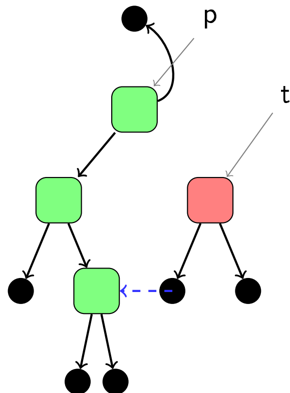
The good one!



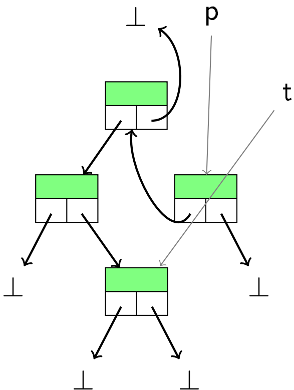


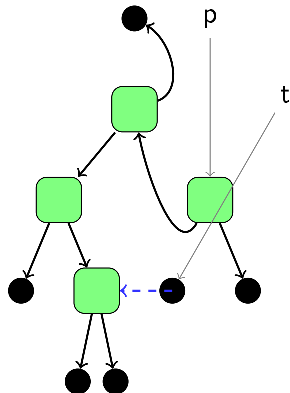
The good one!



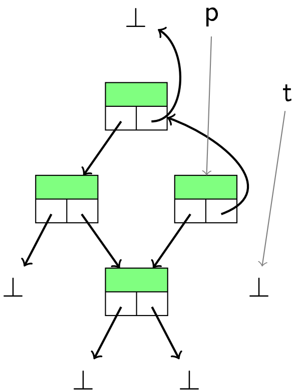


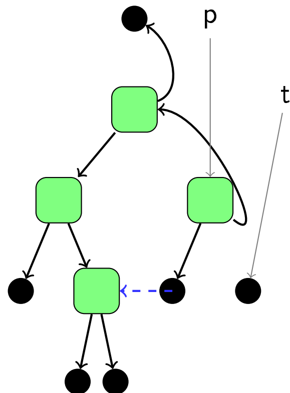
The good one!



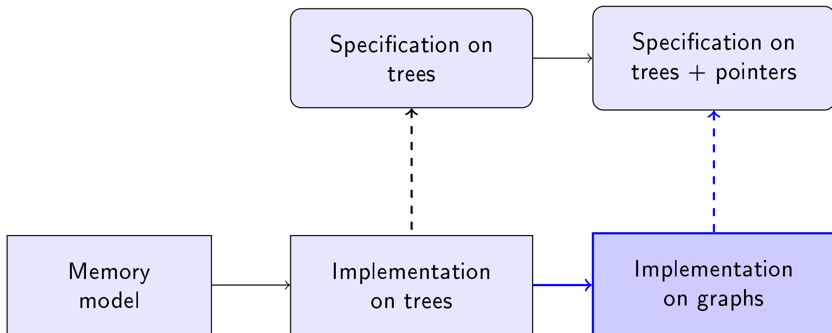


The good one!





Implementation on graphs



Differences with trees

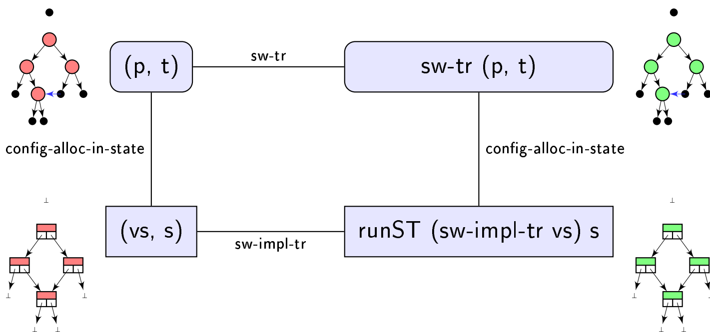
The implementation is exactly the same as the one on trees. We only take into account the additionnal pointers.

from trees ...

consts addr-of :: (('r, 'v) addr tag, 'b) tree ⇒ 'r ref
primrec
  addr-of (Leaf rf) = Null
  addr-of (Node n l r) = Ref (addr-of-tag n)

... to graphs

consts addr-or-ptr ::(('r, 'v) addr tag, 'r ref) tree ⇒ 'r ref
primrec
  addr-or-ptr (Leaf rf) = rf
  addr-or-ptr (Node n l r) = Ref (addr-of-tag n)



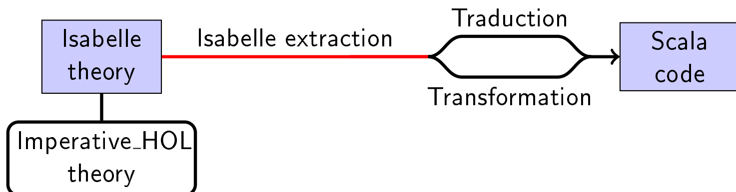
Final theorem

theorem impl-correct-tidied:
⟦ t-marked m t; t-marked-ext t {}; distinct (reach t);
tree-alloc-in-state addr-or-ptr t s ⟧
$\Longrightarrow$ config-alloc-in-state addr-or-ptr
(Leaf Null, mark-all True R t) (runST (sw-impl-tr (Null, addr-or-ptr t)) s)

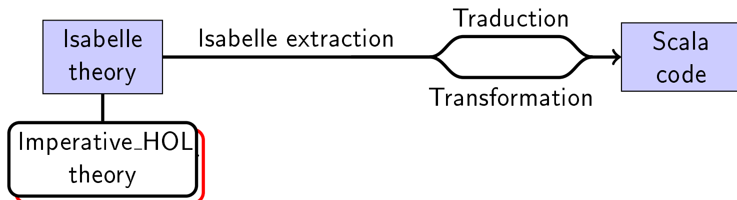
Outline

1. Setting
   - Introduction
   - The Schorr-Waite algorithm
2. Schorr-Waite on trees
   - Definitions
   - Verification
3. Memory model
   - Recall of monads
   - State reader/transformer monads in Isabelle
4. Implementation on trees
   - Definitions
   - Verification
5. Schorr-Waite on trees with pointers
   - Choosing the spanning tree
6. Implementation on graphs
   - Verification
7. Conclusion

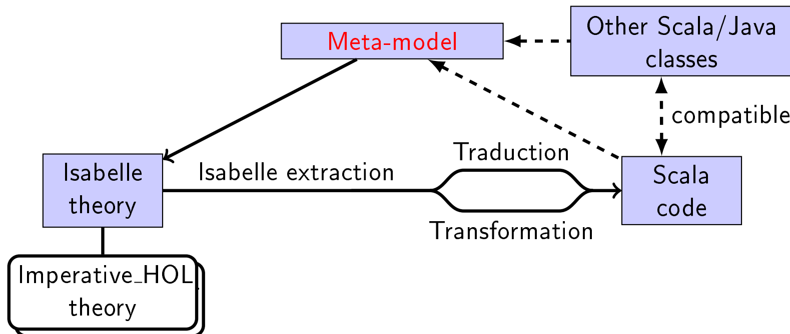
Verification of pointer structures



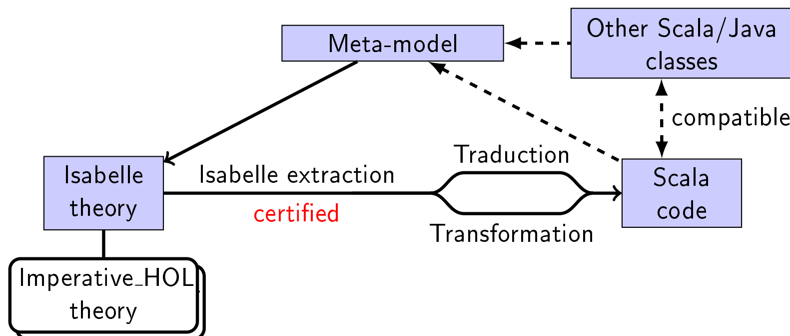
Verification of pointer structures



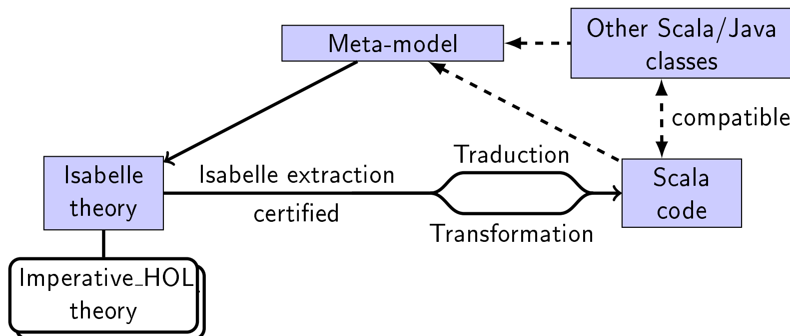
Verification of pointer structures



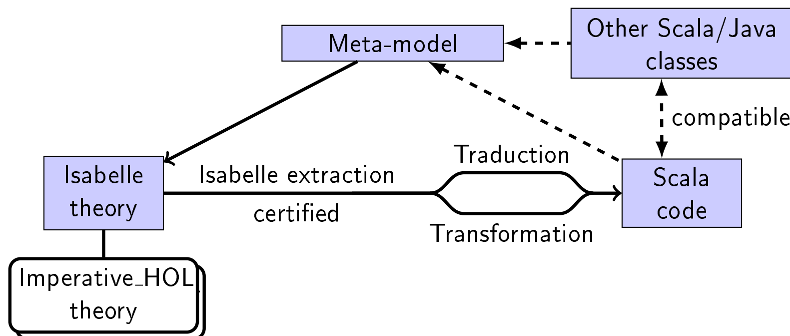
Verification of pointer structures



Verification of pointer structures



Verification of pointer structures



Thank you for your attention

Jean-Raymond Abrial.
Event based sequential program development: Application to constructing a pointer program.
In *Formal Methods Europe (FME)*, LNCS 2805, pages 51–74, 2003.

Richard Bornat.
Proving pointer programs in Hoare logic.
In *Mathematics of Program Construction (MPC)*, LNCS 1837, pages 102–126, 2000.

Thierry Hubert and Claude Marché.
A case study of C source code verification: the Schorr-Waite algorithm.
In *Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2005.

Simon L. Peyton Jones and Philip Wadler.
Imperative functional programming.

In *POPL*, pages 71–84, 1993.

Alexey Loginov, Thomas Reps, and Mooly Sagiv.
Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm.
In *Proc. of SAS-06 Sagiv, M.; Reps, T.; and*, 2006.

Farhad Mehta and Tobias Nipkow.
Proving pointer programs in higher-order logic.
Information and Computation, 199:200–227, 2005.

Eugenio Moggi.
Computational lambda-calculus and monads.
In *LICS*, pages 14–23, 1989.

Eugenio Moggi.
Notions of computation and monads.
Information and Computation, 93(1):55–92, 1991.

Simon Peyton Jones.

Special issue: Haskell 98 language and libraries.
Journal of Functional Programming, 13, January 2003.

H. Schorr and W. Waite.
An efficient machine independent procedure for garbage collection in various list structures.
Communications of the ACM, 10:501–506, 1967.

Philip Wadler.
Comprehending monads.
In *LISP and Functional Programming*, pages 61–78, 1990.