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**
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 interesting 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)
The Schorr-Waite algorithm

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
The Schorr-Waite algorithm

**Steps: Push**

<table>
<thead>
<tr>
<th>p</th>
<th>t</th>
</tr>
</thead>
</table>

```
push
```

```
L
```

```
p
```

```
t
```
The Schorr-Waite algorithm

Steps: Swing

\[ L \rightarrow p \rightarrow R \]

\[ t \rightarrow \text{swing} \rightarrow t \]
The Schorr-Waite algorithm

**Steps:** Pop

```
<table>
<thead>
<tr>
<th>p</th>
<th>R</th>
<th>t</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>p</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>t</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>p</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

The Schorr-Waite algorithm

Demo
Schorr-Waite on trees

Specification on trees

Specification on trees + pointers

Memory model

Implementation on trees

Implementation on graphs
We represent trees with the help of 3 datatypes:

**trees themselves**

```haskell
datatype (`a, `l) tree =
  Leaf `l
  | Node `a ((`a, `l) tree) ((`a, `l) tree)
```

**the 2 directions used to remember the reversed pointer**

```haskell
datatype dir = L | R
```

**the labeled elements of the tree**

```haskell
datatype `a tag = Tag bool dir `a
```
The algorithm

The definition uses 2 auxiliary functions:

**the termination condition**

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

```haskell
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**

```haskell
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 \textbf{sw-tr} :: (('a \text{ tag}, 'l) \text{ tree} \ast ('a \text{ tag}, 'l) \text{ tree})
\Rightarrow (('a \text{ tag}, 'l) \text{ tree} \ast ('a \text{ tag}, 'l) \text{ tree})

\textbf{where} sw-tr \text{ args} = (\text{if} \ (\text{sw-term} \ \text{args}) \ \text{then} \ \text{args} \ \text{else} \ \text{sw-tr} \ (\text{sw-body} \ \text{args}))

which terminates

\textbf{termination} \ sw-tr
\textbf{apply} (relation measures [
\lambda \ (p,t). \ \text{unmarked-count} \ p + \text{unmarked-count} \ t,
\lambda \ (p,t). \ \text{left-count} \ p + \text{left-count} \ t,
\lambda \ (p,t). \ \text{size} \ p])
\textbf{by} \ \text{simp} \ (\text{fastsimp split add: tree.splits tag.splits})
Homogeneity of a tree (property preserved by \(t\))

\[
\text{consts } \text{t-marked} :: \text{bool} \Rightarrow (\text{\textquoteleft a tag, \textquoteleft l}) \text{ tree} \Rightarrow \text{bool} \\
\text{primrec} \\
\text{t-marked } m \ (\text{Leaf } rf) = \text{True} \\
\text{t-marked } m \ (\text{Node } n \ l \ r) = \ (\text{case } n \ of \ (\text{Tag } m\' \ d \ v) \Rightarrow \\
((m \rightarrow (d = R)) \land m' = m \land \text{t-marked } m \ l \land \text{t-marked } m \ r))
\]

Marking a tree

\[
\text{fun } \text{mark-all} :: \text{bool} \Rightarrow \text{dir} \Rightarrow (\text{\textquoteleft a tag, \textquoteleft l}) \text{ tree} \Rightarrow (\text{\textquoteleft a tag, \textquoteleft l}) \text{ tree } \text{where} \\
\text{mark-all } m \ d \ (\text{Leaf } rf) = \text{Leaf } rf \\
| \text{mark-all } m \ d \ (\text{Node } (\text{Tag } - - \ v) \ l \ r) = \\
\quad (\text{Node } (\text{Tag } m \ d \ v) \ (\text{mark-all } m \ d \ l) \ (\text{mark-all } m \ d \ r))
\]

Correction

\[
\text{theorem } \text{sw-tr-correct:} \\
\text{t-marked } m \ t \implies \text{sw-tr } (\text{Leaf } rf, t) = (\text{Leaf } rf, \text{mark-all } \text{True } R \ t)
\]
Memory model

- Specification on trees
- Implementation on trees
- Specification on trees + pointers
- Implementation on graphs

---

Setting Trees Memory Impl/Trees Graphs Impl/Graphs Conclusion
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]**
A monad is composed of:

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

following 3 rules:

- left identity: \( \forall f \, \forall v. \ (return \ v) \, >>= \, f = f \, v \)
- right identity: \( \forall a. \ (a \, >>= \, return) = a \)
- associativity:
  
  \[ \forall x \, f \, g. \ (x \, >>= \, (\lambda v. \, f \, v)) \, >>= \, g = x \, >>= \, (\lambda v. \, (f \, v \, >>= \, g)) \]
Recall of monads

**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:** \( ∀ f. \ (\text{return } v) \ >> \ f = f \ v \)
- **right identity:** \( ∀ a. \ (a \ >> \ \text{return}) = a \)
- **associativity:**
  \[
  ∀ x f g. \ (x \ >> (λ v. f v)) \ >> g = x \ >> (λ v. (f v \ >> g))
  \]
The State-Transformer monad: ('a, 's) ST

\[
\text{avec } x \ 'a, f \ 'a \Rightarrow ('b, 's) ST \text{ et } g \ 'b \Rightarrow ('c, 's) ST,
\]

\[
\text{doST}\{ \\
y \leftarrow f \ x; \\
z \leftarrow g \ y; \\
\text{returnST} \ z;
\}
\]

The state \( s \) is implicitly transmitted to \( f \) and \( g \).

Definition:
- a type constructor:
  - \text{datatype} ('a, 's) ST = ST ('s \Rightarrow ('a \times 's))
- and 2 operators:
  - \text{returnST} a = ST (\lambda s. (a, s))
  - \text{bindST} m f = ST (\lambda s. (\lambda (x, s'). \text{runST} (f x) s') (\text{runST} m s))
The State-Transformer monad: ('a, 's) ST

\[ \text{with } x : 'a, \ f : 'a \Rightarrow ('b, 's) ST \text{ and } g : 'b \Rightarrow ('c, 's) ST, \]

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

The state \( s \) is implicitly transmitted to \( f \) and \( g \).

Definition:
- a type constructor:
  - datatype \((a, s) ST = ST (s \Rightarrow (a \times s))\)
- and 2 operators:
  - returnST \( a = ST (\lambda s. (a, s))\)
  - bindST \( m f = ST (\lambda s. (\lambda (x, s'). runST (f x) s') (runST m s))\)
State reader/transformer monads in Isabelle

('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 \( h : 'd \Rightarrow 'e \):

\[
\begin{align*}
\text{doST } \{ & \quad \text{(with ST)} \\
& \quad \text{let} \\
& \quad \quad \text{va} = \text{runSR } a \ h0; \\
& \quad \quad \text{vga} = \text{runSR } (g \ \text{va}) \ h0; \\
& \quad \quad (x, h1) = \text{runST } (f \ \text{vga} \ b) \ h0 \\
& \quad \quad \text{in} \\
& \quad \quad \quad (h \ x, h1) \\
& \quad \} \\
\end{align*}
\]

and we add syntax:

\[
\begin{align*}
\text{if } \{ \text{c} \} \ \{ \text{a} \} \ \text{else } \{ \text{b} \} & \quad \rightarrow \quad \text{if } \langle \text{doSR} \{ \text{c} \} \rangle \ \text{then } \langle \text{doST} \{ \text{a} \} \rangle \ \text{else } \langle \text{doST} \{ \text{b} \} \rangle \\
\text{[v = v0] while } \{ \text{c} \} \ \{ \text{a} \} & \quad \rightarrow \quad \text{whileST } (\lambda v. \text{doSR} \{ \text{c} \}) (\lambda v. \text{doST} \{ \text{a} \}) v0 \\
\text{while } \{ \text{c} \} \ \{ \text{a} \} & \quad \rightarrow \quad \text{whileST } (\lambda -. \text{doSR} \{ \text{c} \}) (\lambda -. \text{doST} \{ \text{a} \}) ()
\end{align*}
\]
The memory: ('n, 'v) heap

To manipulate references, we add:

**Types**

```plaintext
datatype 'n ref = Ref 'n | Null
record ('n, 'v) heap = heap :: 'n ⇒ 'v
```

**And operators**

```plaintext
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
...```

State reader/transformer monads in Isabelle
Implementation on trees

- Specification on trees
- Specification on trees + pointers
- Implementation on trees
- Implementation on graphs
- Memory model

Implementation on graphs
We define the types:

**of the trees/nodes in the memory**

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

**of a memory containing trees/nodes**

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

**and of tree nodes addresses**

```plaintext
datatype ('r, 'v) addr = Addr 'r 'v
```
Definitions

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) }})
### Definitions

**Implementation**

```plaintext
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**

**Lemma** impl-correct:

\[ (\exists \ m. \ t\text{-marked } m \ t) \land p\text{-marked } p \land \text{distinct } (\text{reach } p \circ \text{reach } t) \land \text{config-alloc-in-state addr-of } (p, t) (vs, s) \]

\[ \implies \text{config-alloc-in-state addr-of } (\text{sw-tr } (p, t)) (\text{runST } (\text{sw-impl-tr } vs) s) \]
Schorr-Waite on trees with pointers

Specification on trees

Memory model

Implementation on trees

Specification on trees + pointers

Implementation on graphs
Choosing the spanning tree

Example: 2 possibility
Choosing the spanning tree

The wrong one!
Choosing the spanning tree

The wrong one!

$$\text{push}$$

$$\text{push}$$
Choosing the spanning tree

The wrong one!
Choosing the spanning tree

The wrong one!
Choosing the spanning tree

The wrong one!

push

pop
Choosing the spanning tree

The good one!
Choosing the spanning tree

The good one!
Choosing the spanning tree

The good one!
Choosing the spanning tree

The good one!

push

push
Choosing the spanning tree

The good one!
Implementation on graphs

- Specification on trees
- Specification on trees + pointers
- Implementation on trees
- Implementation on graphs
- Memory model
Differences with trees

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

**from trees ...**

```plaintext
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**

```plaintext
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:**

\[
\left[ \begin{array}{c}
t \text{-marked } m \ t; \ t \text{-marked-ext } t \ \{ \}; \ \text{distinct (reach } t); \\
\text{tree-alloc-in-state } \text{addr-or-ptr } t \ s
\end{array} \right] \\
\implies \ \text{config-alloc-in-state } \text{addr-or-ptr} \\
( \text{Leaf Null, mark-all True R } t) \ (\text{runST (sw-impl-tr (Null, addr-or-ptr } t)) \ s)
\]
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

- Isabelle theory
- Imperative_HOL theory
- Meta-model
- Isabelle extraction
- Scala code
- Other Scala/Java classes
- Traduction
- Transformation

compatible
Verification of pointer structures

Isabelle theory

Imperative_HOL theory

Meta-model

Isabelle extraction

certified

Traduction

Transformation

Other Scala/Java classes

Scala code

compatible
Verification of pointer structures

Isabelle theory

Meta-model

Isabelle extraction

certified

Traduction

Transformation

Other Scala/Java classes

compatible

Scala code

Imperative_HOL theory
Verification of pointer structures

Isabelle theory

Impérative_HOL theory

Meta-model

Isabelle extraction certified

Traduction Transformation

Other Scala/Java classes compatible

Scala code

Thank you for your attention
Jean-Raymond Abrial.
Event based sequential program development: Application to constructing a pointer program.

Richard Bornat.
Proving pointer programs in Hoare logic.

Thierry Hubert and Claude Marché.
A case study of C source code verification: the Schorr-Waite algorithm.

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

Alexey Loginov, Thomas Reps, and Mooly Sagiv.
Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm.

Farhad Mehta and Tobias Nipkow.
Proving pointer programs in higher-order logic.

Eugenio Moggi.
Computational lambda-calculus and monads.

Eugenio Moggi.
Notions of computation and monads.

Simon Peyton Jones.
Special issue: Haskell 98 language and libraries.

H. Schorr and W. Waite.
An efficient machine independent procedure for garbage collection in various list structures.

Philip Wadler.
Comprehending monads.