Verification of the Schorr-Waite algorithm – From trees to graphs (Extended version)

Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel

IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse

Abstract. This article proposes a method for proving the correctness of graph algorithms by manipulating their spanning trees enriched with additional references. We illustrate this concept with a proof of the correctness of a (pseudo-)imperative version of the Schorr-Waite algorithm by refinement of a functional one working on trees. It is composed of two orthogonal steps of refinement – functional to imperative and tree to graph – finally merged to obtain the result. Our imperative specifications use monadic constructs and syntax sugar, making them close to common imperative languages. This work has been realized within the Isabelle/HOL proof assistant.

Key words: Verification of imperative programs, Pointer algorithms, Program refinement

1 Introduction

The Schorr-Waite algorithm [SW67] is an in-place graph marking algorithm that traverses a graph without building up a stack of nodes visited during traversal. Instead, it codes the backtracking structure within the graph itself while descending into the graph, and restores the original shape on the way back to the root. This makes it particularly space-efficient and therefore appropriate for use in garbage collectors.

A correctness argument for the Schorr-Waite (SW) algorithm is non-trivial, and a large number of correctness proofs, both on paper and machine-assisted, has accumulated over the years. All these approaches have in common that they start from a low-level graph representation, as elements of a heap which are related by pointers (see Section 7 for a discussion).

In this paper, we advocate a development that starts from high-level structures (Section 2), in particular inductively defined trees, and exploits as far as possible the corresponding principles of computation (mostly structural recursion) and reasoning (mostly structural induction). We then proceed by refinement, along two dimensions: on the one hand, by mapping the inductively defined structures to a low-level heap representation (Sections 3 and 4), on the other hand, by adding pointers to the trees, to obtain genuine graphs (Sections 5). These two developments are joined in Section 6.

We argue that this method has several advantages over methods that manipulate lower-level structures:

- Termination of the algorithms becomes easier to prove, as the size of the underlying trees and similar measures can be used in the termination argument.
- Transformation and also preservation of structure is easier to express and to prove than when working on a raw mesh of pointers. In particular, we can state succinctly that the SW algorithm restores the original structure after having traversed and marked it.
- Using structural reasoning such as induction allows a higher degree of proof automation: the prover can apply rewriting which is more directed than the kind of predicate-logic reasoning that arises in a relational representation of pointer structures.

Technically, the main ingredients of our development are, on the higher level, spanning trees with additional pointers parting from the leaf nodes to represent arbitrary (finite) graphs. During the execution of the algorithm, the state space is partitioned into disjoint areas that may only be linked by pointers which satisfy specific invariants. On the lower level, we use state-transformer and state-reader monads for representing imperative programs. The two levels are related by a refinement relation that is preserved during execution of the algorithms.

Even though, taken separately, most of these ingredients are not new (see Section 7 for a discussion of related work), this paper highlights the fact that relatively complex graph algorithms can be dealt with elegantly when perceiving them as refinements of tree algorithms.

The entire development has been carried out in the Isabelle theorem prover [NPW02], which offers a high degree of automation – most proofs are just a few lines long. The formalization itself does not exploit any specificities of Isabelle, but we use Isabelle's syntax definition facilities for devising a readable notation for imperative programs.

2 Schorr-Waite on Pure Trees

A few words on notation before starting the development itself: Isabelle/HOL's syntax is a combination of mathematical notation and ML language. Type variables are written $'a$, $'b$, ..., total functions from α to β are denoted by $\alpha \Rightarrow \beta$ and type constructors are post-fix by default (like $'a\ list$). $\longrightarrow/\Longrightarrow$ are both implication on term-level/meta-level where the meta-level is the domain of proofs. $[\![a_0;\ldots;a_n]\!] \Longrightarrow b$ abbreviates $a_0 \Longrightarrow (\ldots \Longrightarrow (a_n \Longrightarrow a)\ldots)$. Construction and concatenation operators on lists are represented by $x \# xs$ and $xs \mathbin{@} ys$. Sometimes we will judiciously choose the right level of nesting for pattern matching in definitions, in order to take advantage of case splitting to improve automation in proofs.

The high-level version of the algorithm operates on inductively defined trees, whose definition is standard:

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

The SW algorithm requires a tag in each node, consisting of its mark, here represented by a boolean value (*True* for marked, *False* for unmarked) and a "direction" (left or right), telling the algorithm how to backtrack. We store this information as follows:

```
datatype dir = L | R

lemma d-nR-L [simp]: (d ≠ R) = (d = L)
by (case-tac d) simp-all

lemma d-nL-R [simp]: (d ≠ L) = (d = R)
by (case-tac d) simp-all

datatype 'a tag = Tag bool dir 'a
```

We use the accessors *nmark*, *ndir*, *nval* for retrieving the respective components, and we define some simple counting functions

```
consts
  nmark :: 'a tag ⇒ bool
  ndir  :: 'a tag ⇒ dir
  nval  :: 'a tag ⇒ 'a
primrec nmark (Tag m d v) = m
primrec ndir  (Tag m d v) = d
primrec nval  (Tag m d v) = v

consts count :: ('a ⇒ bool) ⇒ ('a,'l) tree ⇒ nat
primrec
  count p (Leaf rf) = 0
  count p (Node v l r) = (if (p v) then 1 else 0) + count p l + count p r

fun unmarked-count :: ('a tag, 'l) tree ⇒ nat
  where unmarked-count t = count (λ n. (¬ nmark n)) t

fun left-count :: ('a tag, 'l) tree ⇒ nat
  where left-count t = count (λ n. (ndir n = L)) t
```

With these preliminaries, we can describe the SW algorithm. It uses two "pointers" t and p (which, for the time being, are trees): t points to the root of the tree to be traversed, and p to the previously visited node. There are three main operations:

- As long as the t node is unmarked, *push* moves t down the left subtree, turns its left pointer upwards and makes p point to the former t node. The latter node is then marked and its direction component set to "left".
- Eventually, the left subtree will have been marked, *i. e.* t's mark is *True*, or t is a Leaf. If p's direction component says "left", the *swing* operation makes

t point to p's right subtree, the roles of p's left and right subtree pointers are inversed, and the direction component is set to "right".
– Finally, if, after the recursive descent, the right subtree is marked and p's direction component says "right", the *pop* operation will make the two pointers move up one level, reestablishing the original shape of t.

The algorithm is supposed to start with an empty p, and it stops if p is empty again and t is marked. The three operations are illustrated in Figure 1 in which dots indicate intermediate steps and leaves are not represented.

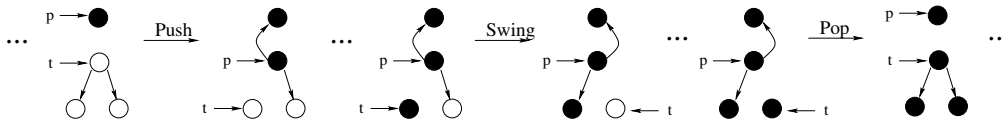

Fig. 1. Operations of the Schorr-Waite algorithm.

Our algorithm uses two auxiliary functions *sw-term* (termination condition) and *sw-body*, the body of the algorithm with three main branches as in the informal characterisation above. The function *sw-body* should not be called if t is marked and p is a Leaf, so it returns an insignificant result in this case.

```
fun sw-term :: (('a tag, 'l) tree * ('a tag, 'l) tree) ⇒ bool where
  sw-term (p, t) = (case p of
      Leaf - ⇒ (case t of Leaf - ⇒ True | (Node (Tag m d v) tlf tr) ⇒ m)
    | -      ⇒ False)

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)
    | - ⇒ (case p of
          Leaf - ⇒ (p, t)
        | (Node (Tag m L v) pl pr) ⇒ ((Node (Tag m R v) t pl), pr)
        | (Node (Tag m R v) pl pr) ⇒ (pr, (Node (Tag m R v) pl t))))
```

The SW algorithm on trees, *sw-tr*, is now easy to define. We note in passing that *sw-tr* is tail recursive. If coding it in a functional programming language, your favorite compiler will most likely convert it to a while loop that traverses the tree without building up a stack.

```
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))
by pat-completeness auto
```

We still have to prove the termination of the algorithm. We note that either the number of unmarked nodes decreases (during *push*), or it remains unchanged

and the number of nodes with "left" direction decreases (during *swing*), or these two numbers remain unchanged and the p tree becomes smaller (during *pop*). This double lexicographic order is expressed in Isabelle as follows (with the predefined function *size*, and functions *unmarked-count* and *left-count* as defined before):

```
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)

declare sw-tr.simps [simp del]
```

Please note that the algorithm works on type (*'a tag*, *'l*) *tree* with an arbitrary type for the data in the leaf nodes, which will later be instantiated by types for references.

For a better understanding of invariants of the algorithm, let's take a look at a typical execution, depicted in Figure 2, with some intermediate steps omitted.

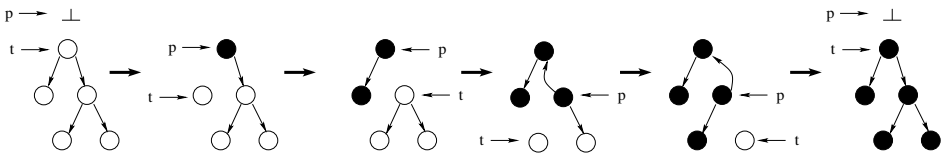

Fig. 2. Typical execution of the Schorr-Waite algorithm

The first thing to note is that the t tree should be consistently marked: Either, it is completely unmarked, or it is completely marked. This is a requirement for the initial tree: A marked root with unmarked nodes hidden below would cause the algorithm to return permaturely, without having explored the whole tree. We sharpen this requirement, by postulating that in a t tree, the direction is "right" iff the node is marked. This is not a strict necessity, but facilitates stating our correctness theorem. We thus arrive at the following two properties *t-marked True* and *t-marked False* for t trees that are defined in one go:

```
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) ⇒
      (((d = R) = m) ∧ m' = m ∧ t-marked m l ∧ t-marked m r))
```

We can similarly state a property of a p tree. We note that such a tree is composed of an upwards branch (the bent arcs in Figure 2) that is again a p-shaped tree, and a downwards branch (the straight lines in Figure 2) that,

depending on the direction, is either a previously marked t tree or an as yet unexplored (and therefore completely unmarked) t tree:

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

Indeed, these two properties are invariants of *sw-body*:

```
lemma t-marked-pres:
  ⟦(p', t') = sw-body (p, t); p-marked p; t-marked m t ⟧ ⟹ (∃ m. t-marked m t')
by (fastsimp split add: tree.splits tag.splits bool.splits dir.splits)

lemma p-marked-pres:
  ⟦(p', t') = sw-body (p, t); p-marked p; t-marked m t⟧ ⟹ p-marked p'
by (simp split add: tree.splits tag.splits bool.splits dir.splits)+

lemma t-marked-pres2:
  ⟦ p-marked p; t-marked m t ⟧ ⟹ ∃ m. t-marked m (snd (sw-body (p, t)))
by (fastsimp split add: tree.splits tag.splits bool.splits dir.splits)

lemma p-marked-pres2:
  ⟦ p-marked p; t-marked m t⟧ ⟹ p-marked (fst (sw-body (p, t)))
by (fastsimp split add: tree.splits tag.splits bool.splits dir.splits)
```

What should the correctness criterion for *sw-tr* be? We would like to state that *sw-tr* behaves like a traditional recursive tree traversal (implicitly using a stack!) that sets the mark to *True*. Unfortunately, SW not only modifies the mark, but also the direction, so the two components have to be taken into account:

```
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 m' d' v) l r) =
    (Node (Tag m d v) (mark-all m d l) (mark-all m d r))
```

By using the function *mark-all* we also capture the fact that the shape of the tree is unaltered after traversal.

Of course, if a tree is consistently marked, it is not modified by marking with *True* and direction "right":

```
lemma t-marked-R-mark-all [rule-format, simp]:
  t-marked True t ⟶ mark-all True R t = t
by (induct t) (auto split add: tag.splits)
```

A key element of the correctness proof is that at each moment of the SW algorithm, given the p and t trees, we can reconstruct the shape of the original

tree (if not its marks) by climbing up the p tree and putting back in place its subtrees:

```
fun reconstruct :: (('a tag, 'l) tree * ('a tag, 'l) tree) ⇒ ('a tag, 'l) tree where
  reconstruct (Leaf rf, t) = t
| reconstruct ((Node n l r), t) =
    (case n of (Tag m d v) ⇒
      (case d of
         L ⇒ reconstruct (l, (Node (Tag m d v) t r))
       | R ⇒ reconstruct (r, (Node (Tag m d v) l t))))
```

For this reason, if two trees t and t' have the same shape (*i. e.* are the same after marking), they are also of the same shape after reconstruction with the same p:

```
lemma mark-all-reconstruct [rule-format]:
  ∀ t t'. mark-all m d t = mark-all m d t' ⟶
  mark-all m d (reconstruct (p, t)) = mark-all m d (reconstruct (p, t'))
by (induct p) (clarsimp split add: tag.splits dir.splits)+
```

Application of *sw-body* does not change the shape of the original tree that p and t are reconstructed to:

```
lemma sw-body-mark-all-reconstruct:
  ⟦ p-marked p; t-marked m' t; ¬ sw-term (p, t) ⟧ ⟹
  mark-all m d (reconstruct (sw-body (p, t))) = mark-all m d (reconstruct (p, t))
by (fastsimp split add: tree.splits tag.splits bool.splits dir.splits
  intro: mark-all-reconstruct)
```

Obviously, if t is *t-marked* and we are in the final state of the recursion (*sw-term* is satisfied), then t is marked as true and p is empty. Together with the invariant of *sw-body* just identified, an induction on the form of the recursion of *sw-tr* gives us:

```
lemma sw-tr-mark-all-reconstruct [rule-format]:
  let (p, t) = args in
  (∀ m. t-marked m t ⟶ p-marked p ⟶
    (let (p', t') = (sw-tr args) in
     mark-all True R (reconstruct (p, t)) = t' ∧ (∃ rf. p' = Leaf rf)))
apply (induct args rule: sw-tr.induct)
apply atomize
apply (subst sw-tr.simps)
apply (clarsimp simp add: split-def Let-def simp del: sw-body.simps)
apply (fastsimp simp add: sw-body-mark-all-reconstruct
  p-marked-pres2 t-marked-pres2    simp del: sw-body.simps
  split add: tree.splits tag.splits)
done
```

For a run of *sw-tr* starting with an empty p, we obtain the desired theorem (which, of course, is only interesting for the non-trivial case m=*False*):

```
theorem sw-tr-correct: t-marked m t
  ⟹ sw-tr (Leaf rf, t) = (p', t')
```

$\Longrightarrow t' = \mathit{mark\text{-}all}\ \mathit{True}\ R\ t \wedge (\exists\ \mathit{rf}.\ p' = \mathit{Leaf}\ \mathit{rf})$
by (*insert sw-tr-mark-all-reconstruct* [*of* (*Leaf rf*, *t*)]) *fastsimp*

To show the brevity of the development, we have reproduced the entire Isabelle script up to this point (we will be less exhaustive in the following).

3 Imperative Language and its Memory Model

This section presents a way to manipulate low-level programs. We use a heap-transformer monad providing means to reason about monadic/imperative code along with a nice syntax, and that should allow to generate similar executable code.

The theory Imperative_HOL [BKH+08] discussed in Section 7 already implements such a monad, however our development started independently of it and we have then used it to improve our version, without generation/extraction for the moment.

3.1 The State Transformer Monad

In this section we define the state-reader and state-transformer monads and a syntax seamlessly mixing them. We encapsulate them in the *SR* – respectively *ST* – datatypes, as functions from a state to a return value – respectively a pair of return value and state.

We can escape from these datatypes with the *runSR* – respectively *runST* and *evalST* – functions which are intended to be used only in logical parts (theorems and proofs) and that should not be extractible.

datatype $('a, 's)\ SR = SR\ 's \Rightarrow 'a$
datatype $('a, 's)\ ST = ST\ 's \Rightarrow 'a \times 's$

consts
$\mathit{runSR} :: ('a, 's)\ SR \Rightarrow 's \Rightarrow 'a$
$\mathit{runST} :: ('a, 's)\ ST \Rightarrow 's \Rightarrow 'a \times 's$

primrec $\mathit{runSR}\ (SR\ m) = m$
primrec $\mathit{runST}\ (ST\ m) = m$

abbreviation $\mathit{evalST} :: ('a, 's)\ ST \Rightarrow 's \Rightarrow 'a$
where $\mathit{evalST}\ \mathit{fm}\ s == \mathit{fst}\ (\mathit{runST}\ \mathit{fm}\ s)$

The *return* (also called *unit*) and *bind* functions for manipulating the monads are then defined classicaly with the infix notations $\trianglerighteq_{SR}$ and $\trianglerighteq_{ST}$ for *bind*s. We add also the function *SRtoST* translating state-reader monads to state-transformer monads and the function *thenST* (with infix notation $\triangleright_{ST}$) abbreviating binding without value transfer.

consts

$returnSR :: 'a \Rightarrow ('a, 's)\ SR$
$returnST :: 'a \Rightarrow ('a, 's)\ ST$
$bindSR :: ('a, 's)\ SR \Rightarrow ('a \Rightarrow ('b, 's)\ SR) \Rightarrow ('b, 's)\ SR$ (**infixr** $\trianglerighteq_{SR}$)
$bindST :: ('a, 's)\ ST \Rightarrow ('a \Rightarrow ('b, 's)\ ST) \Rightarrow ('b, 's)\ ST$ (**infixr** $\trianglerighteq_{ST}$)
$SRtoST :: ('a,'s)\ SR \Rightarrow ('a, 's)\ ST$

defs
$returnSR\ a == SR\ (\lambda\ s.\ a)$
$returnST\ a == ST\ (\lambda\ s.\ (a, s))$
$bindSR\ m\ f == SR\ (\lambda\ s.\ (\lambda\ x.\ \ runSR\ (f\ x)\ s)\ \ (runSR\ m\ s))$
$bindST\ m\ f == ST\ (\lambda\ s.\ (\lambda\ (x, s').\ runST\ (f\ x)\ s')\ (runST\ m\ s))$
$SRtoST\ sr\ == ST\ (\lambda\ s.\ (runSR\ sr\ s, s))$

abbreviation
$thenST :: ('a, 's)\ ST \Rightarrow ('b, 's)\ ST \Rightarrow ('b, 's)\ ST$ (**infixr** $\triangleright_{ST}$ *55*)
where $a \triangleright_{ST} b == a \trianglerighteq_{ST} (\lambda\ \text{-}.\ b)$

We can then verify the monad laws:

lemma *monadSRlaws* :
$\forall\ v\ f.\ (returnSR\ v) \trianglerighteq_{SR} f = f\ v$
$\forall\ a.\ a \trianglerighteq_{SR} returnSR = a$
$\forall\ (x::('s, 'a)\ SR)\ f\ g.\ (x \trianglerighteq_{SR} f) \trianglerighteq_{SR} g = x \trianglerighteq_{SR} (\lambda\ v.\ ((f\ v) \trianglerighteq_{SR} g))$
by(*simp-all add*: *expand-SR-eq SR-run0*)

lemma *monadSTlaws* :
$\forall\ v\ f.\ (returnST\ v) \trianglerighteq_{ST} f = f\ v$
$\forall\ a.\ a \trianglerighteq_{ST} returnST = a$
$\forall\ (x::('a, 's)\ ST)\ f\ g.\ (x \trianglerighteq_{ST} f) \trianglerighteq_{ST} g = x \trianglerighteq_{ST} (\lambda\ v.\ ((f\ v) \trianglerighteq_{ST} g))$
by (*simp-all add*: *expand-ST-eq ST-run0 split*:*prod.splits*)

We define also syntax translations to use the Haskell-like *do*-notation.

The principal difference between the Haskell *do*-notation and this one is the use of state-readers for which order does not matter. With some syntax transformations, we can simply compose several state readers into one as well as give them as arguments to state writers, almost as it is done in imperative languages (for which state is the heap). In an adapted context – *i. e.* in $doSR\{\dots\}$ or $doST\{\dots\}$ – we can so use state readers in place of expressions by simply putting them in $\langle\dots\rangle$, the current state being automatically provided to them, only thanks to the syntax transformation which propagates the same state to all $\langle\dots\rangle$. We also add syntax for *let* ($letST\ x = a^{SR}; b^{ST}$) and *if* ($if\ (a^{SR})\ \{b^{ST}\}\ else\ \{c^{ST}\}$).

We describe this syntax with a rewriting system, applied bottom-up – that is also the procedure applied by the Isabelle parser. This means that arguments of the rule are in normal form when the rule is applied.

We underscore purely syntactic functions:

doSTSR which transforms a state-reader returning a state-transformer – of type $(('a, 's)\ ST, 's)\ SR$ – into a state-transformer – of type $('a, 's)\ ST$

doSR0 which takes a variable s representing the state as argument and replaces all $\langle\ e\ \rangle$ by $runST\ e\ s$ in an expression.

$$\begin{array}{lll}
\underline{doSTSR}\ p & \equiv & ST\ (\lambda s.\ runST\ (\underline{doSR0}\ s\ p)\ s) \\
\underline{doSR0}\ s\ p & \equiv & p[\ runSR\ (\underline{doSR0}\ s\ x)\ s\ /\ \langle\ x\ \rangle\ ]
\end{array}$$

$$\begin{array}{ll}
doST\ \{\ x \leftarrow p;\ q\ \} & \longrightarrow (\underline{doSTSR}\ p) \trianglerighteq_{ST} (\lambda x.\ doST\ \{\ q\ \}) \\
doST\ \{\ p;\ q\ \} & \longrightarrow (\underline{doSTSR}\ p) \triangleright_{ST} doST\ \{\ q\ \} \\
doST\ \{\ p\ \} & \longrightarrow \underline{doSTSR}\ p \\
doST\ \{\ letST\ x = p;\ q\ \} & \longrightarrow doST\ \{\ x \leftarrow SRtoST\ (doSR\ \{\ p\ \});\ q\ \}
\end{array}$$

$$doSR\ \{\ p\ \} \longrightarrow SR\ (\lambda s.\ \underline{doSR0}\ s\ p)$$

For example with $f^{'a \Rightarrow ('b,\ 's)\ ST}$, $a^{('a,'s)\ SR}$, $g^{((),\ 's)\ ST}$ and $h^{'b \Rightarrow 'd}$, all these expressions are equivalent:

- $doST\ \{\ x \leftarrow f\ \langle a\rangle;\ g;\ returnST\ (h\ x)\}$
- $doST\ \{\ va \leftarrow SRtoST\ a;\ x \leftarrow f\ va;\ g;\ returnST\ (h\ x)\}$
- $doST\ \{\ x \leftarrow ST\ (\lambda s.\ runST\ (f\ (runSR\ a\ s))\ s);\ g;\ returnST\ (h\ x)\}$
- $ST\ (\lambda s.\ runST\ (f\ (runSR\ a\ s))\ s) \trianglerighteq (\lambda x.\ g \triangleright returnST\ (h\ x))$

We also add syntax for *if*:

$doST\ \{if\ (c)\ \{a\}\ else\ \{b\}\} \longrightarrow doST\ \{if\ \langle doSR\{c\}\rangle\ then\ (doST\{a\})\ else\ (doST\{b\})\}$

and we define the *whileST* combinator, inspired from the *while* combinator definition of the Isabelle library.

As explained in the Isabelle function tutorial, in general, termination of functions has to be proved to give access to unconditional simplification rules. This is because functions must be total to prevent inconsistencies. However, in the particular case of tail-recursive functions, it is not necessary because there is a total function satisfying them, even if they are non-terminating.

We can thus define the *whileST* combinator by simply adding the *tailrec* option to the function definition.

The only difference with the *while* combinator, is the encapsulation in monads.

```
function (tailrec)
  mwhile :: ('v ⇒ (bool, 's) SR) ⇒ ('v ⇒ ('v, 's) ST) ⇒ 'v ⇒ 's ⇒ ('v × 's)
where
  mwhile-unfold[simp del]: mwhile b c v s =
   (if runSR (b v) s
     then (case (runST (c v) s) of ( v', s') ⇒ mwhile b c v' s')
     else (v, s))
constdefs whileST :: ('v ⇒ (bool, 's) SR) ⇒ ('v ⇒ ('v, 's) ST) ⇒ 'v ⇒ ('v, 's) ST
  where whileST b c v == ST (mwhile b c v)
```

We finally add syntax for *whileST*:

$[v = v0]$ *while* (c) $\{a\} \longrightarrow$ *whileST* $(\lambda v.\ doSR\{c\})$ $(\lambda v.\ doST\{a\})$ *v0*
while (c) $\{a\} \longrightarrow$ *whileST* $(\lambda\text{-}.\ doSR\{c\})$ $(\lambda\text{-}.\ doST\{a\})$ () and
we can compare the two definitions: *whileST b c v* = (*doST*{*if*(⟨*b v*⟩){*v'* ← *c v*; *whileST b c v'*}*else*{*returnST v*}})
while b c v = (*if b v then while b c* (*c v*) *else v*)

3.2 The Heap Transformer Monad

We define a heap we will use as the state in the state-reader/transformer monads. We represent it by an extensible record containing a field for the values.

As the Schorr-Waite algorithm doesn't need allocation of new references, our heap simply is a total function from references to values. (We use a record here because of developments already under way and needing further components.)

record (*'n*, *'v*) *heap* = *heap* :: *'n* ⇒ *'v*
abbreviation *heap-upd s n v* == *s*(|*heap* := (*heap s*)(*n* := *v*)|)

We assume that we have a data type of references, which can either be *Null* or point to a defined location:

datatype *'n ref* = *Ref 'n* | *Null*

To read and write the heap, we define the corresponding primitives *read* and *write*. To access directly to the fields of structures in the heap, we also add the *get* $(a{\cdot}b)$, *rget* $(r \to b)$ and *rupdate* $(r \to b := v)$ operators, taking an accessor (b) as argument.

4 Implementation for Pure Trees

In this section, we provide a low-level representation of trees as structures connected by pointers that are manipulated by an imperative program. This is the typical representation in programming languages like C, and it is also used in most correctness proofs of SW.

4.1 Data Structures

These structures are either empty (corresponding to a leaf with a null pointer, as we will see later) or nodes with references to the left and right subtree:

datatype (*'a*, *'r*) *struct* = *EmptyS* | *Struct 'a* (*'r ref*) (*'r ref*)

We define then accessors $v (value) $l (left) and $r (right) for the (*'a*, *'r*) *struct* datatype, and accessors $*mark*, $*dir* and $*val* for the *'a tag* datatype.

consts

```
  struct-v :: ('a, 'r) struct ⇒ 'a ⇒ ('a, 'r) struct        ($v)
  struct-l :: ('a, 'r) struct ⇒ 'r ref ⇒ ('a, 'r) struct    ($l)
  struct-r :: ('a, 'r) struct ⇒ 'r ref ⇒ ('a, 'r) struct    ($r)

primrec
  struct-v (Struct v l r) v' = (Struct v' l r)
  struct-v EmptyS v' = EmptyS
primrec
  struct-l (Struct v l r) l' = (Struct v l' r)
  struct-l EmptyS l' = EmptyS
primrec
  struct-r (Struct v l r) r' = (Struct v l r')
  struct-r EmptyS r' = EmptyS
```

and at the same time, accessors for the *'a tag* datatype defined in Section 2.

```
consts
  acc-mark :: 'a tag ⇒ bool ⇒ 'a tag    ($mark)
  acc-dir :: 'a tag ⇒ dir ⇒ 'a tag      ($dir)
  acc-val :: 'a tag ⇒ 'a ⇒ 'a tag       ($val)

primrec acc-mark (Tag m d v) m' = (Tag m' d  v )
primrec acc-dir (Tag m d v) d'  = (Tag m  d' v )
primrec acc-val (Tag m d v) v'  = (Tag m  d  v')
```

Traditionally, in language semantics, the memory is divided into a heap and a stack, where the latter contains the variables. In our particular case, we choose a greatly simplified representation, because we just have to accommodate the variables pointing to the trees p and t. Our heap will be a type abbreviation for heaps whose values are structures:

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

4.2 An Imperative Algorithm

We now have an idea of the low-level memory representation of trees and can start divising an imperative program that manipulates them (as we will see, with a similar outcome as the high-level program of Section 2). The program is a while loop, written in monadic style, that has as one main ingredient a termination condition:

```
constdefs
  sw-impl-term :: ('r ref × 'r ref) ⇒ ('r, 'v tag) str-heap SR
  sw-impl-term vs == doSR { (case vs of (ref-p, ref-t) ⇒
   (case ref-p of
      Null ⇒ (case ref-t of Null ⇒ True | t ⇒ ((⟨ read t ⟩·$v)·$mark) )
    | - ⇒ False))}
```

The second main ingredient of the while loop is its body:

```
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 (case t of Null ⇒ True | - ⇒ (⟨read t⟩·$v)·$mark) {
    (if (⟨ p → ($v oo $dir) ⟩ = L) {   (** swing **)
        letST rt = ⟨p → $r⟩;
        p → $r := ⟨p → $l⟩;
        p → $l := t;
        p → ($v oo $dir) := R;
        returnST (p, rt)
    } else {                    (** pop **)
        letST rp = ⟨p → $r⟩;
        p → $r := t;
        returnST (rp, p)
    })
  } else {                      (** push **)
    letST rt = ⟨t → $l⟩;
    t → $l := p;
    t → ($v oo $mark) := True;
    t → ($v oo $dir) := L;
    returnST (t, rt) }
  })
```

The termination condition and loop body are combined in the following imperative program:

```
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 }})
```

4.3 Correctness

Before we can describe the implementation of inductively defined trees in low-level memory, let us note that we need to have a means of expressing which node of a tree is mapped to which memory location. For this, we need trees adorned with address information:

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

For later use, we also introduce some accessors:

```
primrec addr-of-tag :: ('r, 'v) addr tag ⇒ 'r
where addr-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ ref)
primrec val-of-tag :: ('r, 'v) addr tag ⇒ 'v
where val-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ v)
```

We can now turn to characterizing the implementation relation of trees in memory, which we define gradually, starting with a relation which expresses that a (non-empty) node n with subtrees l and r is represented in state s. Remember that node n contains its address in memory. It is not possible that the structure at this address is empty. We therefore find a structure with a field corresponding to the value of n (just remove the address, which is not represented in the structure) and left and right pointers to the l and r subtrees:

```
primrec val-proj-of-tag :: ('r, 'v) addr tag ⇒ 'v tag
where val-proj-of-tag (Tag m d av) = (case av of (Addr ref v) ⇒ (Tag m d v))

constdefs struct-alloc-in-state ::
  ('t ⇒ 'r ref) ⇒ ('r, 'v) addr tag ⇒ 't ⇒ 't ⇒ ('r,'v tag) str-heap ⇒ bool
  struct-alloc-in-state ac n l r s ==
   (case (heap s (addr-of-tag n)) of
      EmptyS ⇒ False
    | Struct ns ref-l ref-r ⇒ ns = val-proj-of-tag n ∧ ref-l = ac l ∧ ref-r = ac r)
```

Please ignore the accessor parameter *ac* for the moment. We will instantiate it with different functions, depending on whether we are working on trees (this section) or on graphs (in Section 6).

Given the representation of a node in memory, we can define the representation of a tree: Just traverse the tree recursively and check that each node is correctly represented:

```
consts tree-alloc-in-state :: ((('r, 'v) addr tag, 'l) tree ⇒ 'r ref) ⇒
  (('r, 'v) addr tag, 'l) tree ⇒ ('r,'v tag) str-heap ⇒ bool
primrec
  tree-alloc-in-state ac (Leaf rf) s = True
  tree-alloc-in-state ac (Node n l r) s =
     (struct-alloc-in-state ac n l r s
    ∧ tree-alloc-in-state ac l s ∧ tree-alloc-in-state ac r s)
```

Finally, a configuration (the p and t trees) is correctly represented if each of the trees is, and the p variable contains a reference to the p tree, and similarly for t:

```
constdefs config-alloc-in-state :: ((('r, 'v) addr tag, 'l) tree ⇒ 'r ref) ⇒
   (('r, 'v) addr tag, 'l) tree ∗ (('r, 'v) addr tag, 'l) tree ⇒
    ('r ref × 'r ref) × ('r, 'v tag) str-heap ⇒ bool where
  config-alloc-in-state ac vs rs ==
    (let ((p,t),((ref-p, ref-t),s)) = (vs, rs) in
      tree-alloc-in-state ac p s ∧ tree-alloc-in-state ac t s ∧ ref-p = ac p ∧ ref-t = ac t)
```

Let us now present the first intended instantiation of the *ac* parameter appearing in the above definitions: It is a function that returns the address of a non-empty node, and always *Null* for a leaf:

```
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)
```

We can now state our first result: for a couple of p and t trees correctly allocated in a state s, the low-level and high-level algorithms have the same termination behaviour:

```
lemma sw-impl-term-sw-term:
config-alloc-in-state addr-of pt (ref-pt, s)
```

$\Longrightarrow$ *runSR* (*sw-impl-term ref-pt*) *s* = *sw-term pt*

Before discussing the correctness proof, let us remark that the references occurring in the trees have to be unique. Otherwise, the representation of a tree in memory might not be a tree any more, but might contain loops or joint subtrees. Given the list of references occuring in a tree:

consts *reach* :: (('*r*, '*v*) *addr tag*, '*l*) *tree* ⇒ '*r list*
primrec
reach (*Leaf rf*) = []
reach (*Node n l r*) = (*addr-of-tag n*) # (*reach l*) @ (*reach r*)

we will in the following require the p and t trees to have disjoint ("distinct") reference lists.

The correctness argument of the imperative algorithm is now given in the form of a simulation theorem: A computation with *sw-tr* carried out on trees p and t and producing trees p' and t' can be simulated by a computation with *sw-impl-tr* starting in a state implementing p and t, and ending in a state implementing p' and t'.

The proof is by induction on the structure of *sw-tr*, and the induction step requires *sw-body* and *sw-impl-body* to proceed in lockstep.

lemma *sw-impl-body-config-alloc*:
⟦ *config-alloc-in-state addr-of* (*p*, *t*) ((*ref-p*, *ref-t*), *s*); *t-marked m t*; *p-marked p*;
¬ *runSR* (*sw-impl-term* (*ref-p*, *ref-t*)) *s*; *distinct* (*reach p* @ *reach t*) ⟧
$\Longrightarrow$ *config-alloc-in-state addr-of*
(*sw-body* (*p*, *t*)) (*runST* (*sw-impl-body* (*ref-p*, *ref-t*)) *s*)

Of course, the preconditions of this lemma have to remain invariant as well. We have already proved this for *t-marked* and *p-marked* in Section 2, and we can also show invariance of the property separating p and t:

lemma *sw-impl-body-distinct* [*rule-format*]:
⟦ (*p'*, *t'*) = *sw-body* (*p*, *t*); *t-marked m t*; *p-marked p*;
distinct (*reach p* @ *reach t*) ⟧ $\Longrightarrow$ *distinct* (*reach p'* @ *reach t'*)

The proofs of both lemmas go through without much ado: essentially a single `fastsimp` command in Isabelle. For proving the correctness theorem for the entire implementation, we have derived an easier to handle induction principle:

lemma *sw-tr-induct-rule-rel* [*rule-format*]:
assumes *invariant*:!!*a vs s*. *P a* (*vs*,*s*) ⟶ ¬ *sw-term a*
⟶ *P* (*sw-body a*) (*runST* (*sw-impl-body vs*) *s*)
and *terminate*: !!*a s*. *P a s* ⟶ *sw-term a* ⟶ *Q a s*
and *config*: !! *a s*. *P a s* $\Longrightarrow$ *config-alloc-in-state addr-of a s*
shows ∀ *vs s*. *P a* (*vs*,*s*) ⟶ *Q* (*sw-tr a*) (*runST* (*sw-impl-tr vs*) *s*)

and apply it in the proof of the theorem:

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*)

lemma [*simp*]: *obj* $\neq$ *EmptyS* $\Longrightarrow$ (($l *obj v*)·$l) = *v*
by *simp*

5 Schorr-Waite on Trees with Pointers

The Schorr-Waite algorithm has originally been conceived for genuine graphs, and not for trees. We will now add "pointers" to our trees to obtain a representation of a graph as a spanning tree with additional pointers. This is readily done, by instantiating the type variables of the leaves to the type of references. Thus, a leaf can now represent a null pointer (*Leaf Null*), or a reference to *r*, of the form *Leaf* (*Ref r*).

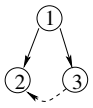

Fig. 3. A tree with additional pointers (drawn as dashed lines)

For example, the graph of Figure 3 could be represented by the following tree, with references of type *nat*:

Node (*Tag False L* (*Addr 1* ()))
(*Node* (*Tag False L* (*Addr 2* ())) (*Leaf Null*) (*Leaf Null*))
(*Node* (*Tag False L* (*Addr 3* ())) (*Leaf* (*Ref 2*)) (*Leaf Null*))

A given graph might be represented by different spanning trees with additional pointers, but the choice is not indifferent: it is important that the graph is represented by a spanning tree that has an appropriate form, so that the low-level algorithm of Section 4 starts backtracking at the right moment. To characterize this form and to get an intuition for the simulation proof presented in Section 6, let's take a look at a "good" (Figure 4) and a "bad" spanning tree (Figure 5).

In Figure 4, the decisive step occurs when *p* points to node 3. Since the high-level algorithm proceeds structurally, it does not follow the additional pointer. Its *t* tree will therefore be a leaf, and the algorithm will start backtracking at this moment. The low-level representation does not distinguish between pointers to subtrees and additional pointers in leaf nodes, so that the *t* pointer will follow the link to node 4, just to discover that this node is already marked. For this reason, also the low-level algorithm will start backtracking.

The situation is different in Figure 5, when *p* reaches node 2. The high-level algorithm finishes its recursive descent at this point and starts backtracking,

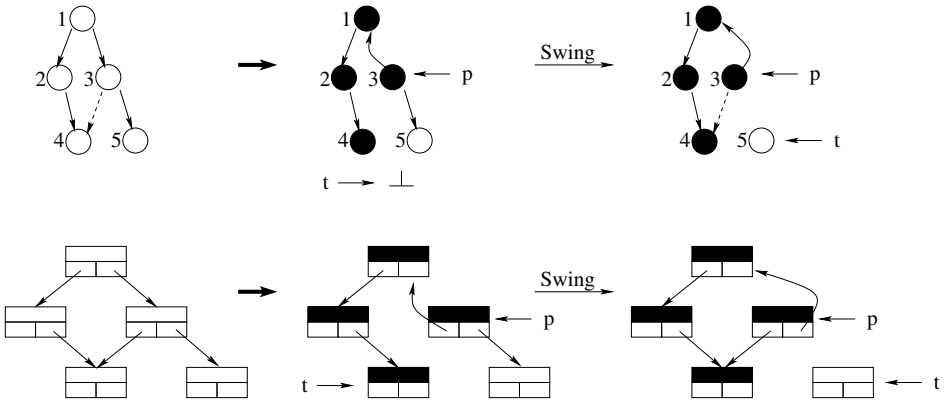

Fig. 4. Low-level graph with a "good" spanning tree

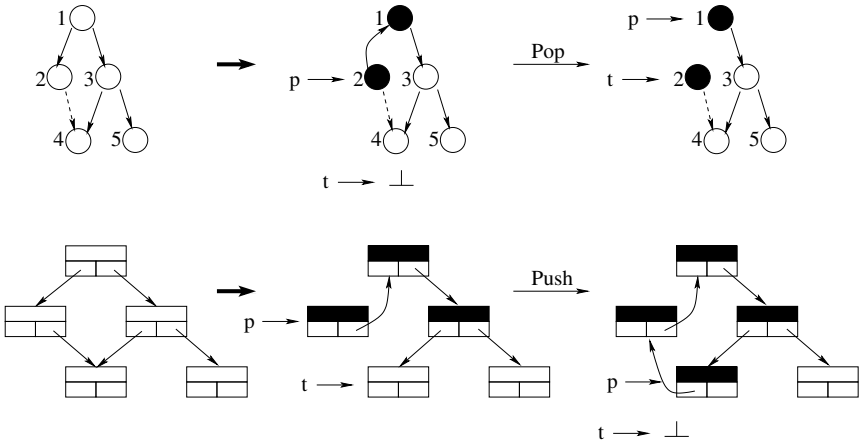

Fig. 5. The same graph with a bad spanning tree

leaving node 4 unmarked. However, the low-level version proceeds to node 4 with its t pointer, marks it and starts exploring its subtrees. At this point, the high-level and low-level versions of the algorithm start diverging irrecoverably.

What then, more generally, is a "good" spanning tree? It is one where all additional pointers reference nodes that have already been visited before, in a pre-order traversal of the tree. This way, we can be sure that by following such an additional pointer, the low-level algorithm discovers a marked node and returns. We formalize this property by a predicate that traverses a tree recursively and checks that additional pointers only point to a set of allowed external references *extrefs*. When descending into the left subtree, we add the root to this set, and when descending into the right subtree, the root and the nodes of the left subtree.

consts

t-marked-ext :: $((\prime r, \prime v)\ addr\ tag, \prime r\ ref)\ tree \Rightarrow \prime r\ set \Rightarrow bool$

```
primrec
  t-marked-ext (Leaf rf) extrefs = (case rf of Null ⇒ True | Ref r ⇒ r ∈ extrefs)
  t-marked-ext (Node n l r) extrefs =
        (t-marked-ext l (insert (addr-of-tag n) extrefs)
      ∧ t-marked-ext r ((insert (addr-of-tag n) extrefs) ∪ set (reach l)))
```

There is a similar property *p-marked-ext*, less inspiring, for p trees, which we give here for completeness:

```
consts
  reach-visited :: (('r, 'v) addr tag, 'r ref) tree ⇒ 'r list
primrec
  reach-visited (Leaf rf) = []
  reach-visited (Node n l r) = (addr-of-tag n) #
    (case n of  (Tag m d v) ⇒
    (case d of
       L ⇒ reach-visited l
     | R ⇒ reach l @ reach-visited r))
consts
  p-marked-ext :: (('r, 'v) addr tag, 'r ref) tree ⇒ 'r set ⇒ bool
primrec
  p-marked-ext (Leaf rf) extrefs = (rf = Null)
  p-marked-ext (Node n l r) extrefs =
    (case n of  (Tag m d v) ⇒
    (case d of
       L ⇒ (p-marked-ext l ((insert (addr-of-tag n) extrefs) ∪ set (reach r))
            ∧ t-marked-ext r ((insert (addr-of-tag n) extrefs) ∪ set (reach-visited l)))
     | R ⇒ (t-marked-ext l (insert (addr-of-tag n) (set (reach-visited r)))
            ∧ p-marked-ext r ((insert (addr-of-tag n) extrefs) ∪ set (reach l)))))
```

Let us insist on this point, because it is essential for the method advocated in this paper: Intuitively, we partition the state space into disjoint areas of addresses with their spanning trees, as depicted in Figure 6. The properties of these areas are described by the predicates *t-marked-ext* and *p-marked-ext*. Pointers may reach from one area into another area (the *extrefs* of the predicates). For the proof, it will be necessary to identify characteristic properties of these external references.

One of these is that all external references of a t tree only point to nodes marked as true. To prepare the ground, the following function gives us all the references of nodes that have a given mark:

```
consts marked-as-in :: bool ⇒ (('r, 'v) addr tag, 'l) tree ⇒ 'r set
primrec
  marked-as-in m (Leaf rf) = {}
  marked-as-in m (Node n l r) =
     (if (nmark n) = m then {(addr-of-tag n)} else {})
   ∪ marked-as-in m l ∪ marked-as-in m r
```

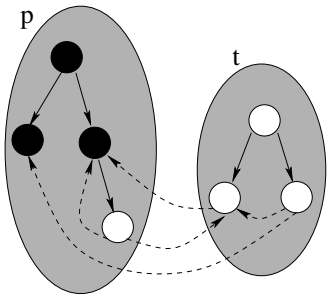

Fig. 6. Partitioning of trees

We can now show some invariants of function *sw-body* that will be instrumental for the correctness proof of Section 6. For example, if all external references of t only point to marked nodes of p, then this will also be the case after execution of *sw-body*:

lemma *marked-ext-pres-t-marked-ext*:
⟦ $(p', t') =$ *sw-body* (p, t); $\neg$ *sw-term* (p, t);
distinct (*reach* p @ *reach* t);
p-marked-ext p (*set* (*reach* t));
t-marked-ext t (*set* (*reach-visited* p));
t-marked m t;
p-marked p;
(*set* (*reach-visited* p)) $\subseteq$ *marked-as-in True* p⟧
$\Longrightarrow$ *t-marked-ext* t' (*set* (*reach-visited* p'))

To conclude this section, let us remark that the refinement of trees by instantiation of the leaf node type described here allows us to state some more invariants, but of course, the correctness properties of the original algorithm of Section 2 remain valid.

6 Implementation for Trees with Pointers

More surprisingly, we will now see that the low-level traversal algorithm of Section 4 also works for all graphs - without the slightest modification of the algorithm! The main justification has already been given in the previous section: There is essentially only one situation when the "pure tree" version and the "tree with pointers" version of the algorithm differ:

- In the high-level "pure tree" version, after a sequence of *push* operations, the t tree will become a leaf. This will be the case exactly when in the low-level version of the algorithm, the correspoding t pointer will become *Null*.
- In the high-level "tree with pointers" version, after some while, the t tree will also become a leaf, but in the low-level version, the t pointer might have moved on to a non-*Null* node. If the underlying spanning tree is well-formed

(in the sense of *t-marked-ext*), the t will point to a marked node, so that the algorithm initiates backtracking in both cases.

We achieve this by taking into account the information contained in leaf nodes. Instead of the function *addr-of* of Section 4, we now parameterize our development with the function *addr-or-ptr* that also returns the references contained in leaf nodes:

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*)

We can now prove an extension of the lemma *sw-impl-term-sw-term* of Section 4.3 establishing the correspondence of the high-level and low-level termination conditions described above. What is new are additional preconditions that we have motivated in Section 5: the p tree is *p-marked-ext*, with external references pointing to the t tree, and inversely for the t tree:

lemma *sw-impl-term-sw-term-ext*:
⟦ *config-alloc-in-state addr-or-ptr* (*p*, *t*) (*ref-pt*, *s*);
p-marked-ext p (*set* (*reach t*));
t-marked-ext t (*set* (*reach-visited p*)) ⟧
⟹ *runSR* (*sw-impl-term ref-pt*) *s* = *sw-term* (*p*, *t*)

The proof for the simulation lemma now proceeds essentially along the same lines as the proof seen for lemma *sw-impl-body-config-alloc*.

lemma *sw-impl-body-config-alloc-ext*:
⟦ *config-alloc-in-state addr-or-ptr* (*p*, *t*) (*vs*, *s*);
t-marked m t; *p-marked p*; ¬ (*runSR* (*sw-impl-term vs*) *s*);
distinct (*reach p* @ *reach t*);

p-marked-ext p (*set* (*reach t*));
t-marked-ext t (*set* (*reach-visited p*));
(*set* (*reach-visited p*)) ⊆ *marked-as-in True p* ⟧
⟹
config-alloc-in-state addr-or-ptr (*sw-body* (*p*, *t*)) (*runST* (*sw-impl-body vs*) *s*)

Now, the proof proceeds along the lines of the proof discussed in Section 4, yielding finally a preservation theorem for configurations:

lemma *impl-correct-ext*:
(∃ *m*. *t-marked m t*) ∧ *p-marked p* ∧ *distinct* (*reach p* @ *reach t*)
∧ *p-marked-ext p* (*set* (*reach t*)) ∧ *t-marked-ext t* (*set* (*reach-visited p*))
∧ (*set* (*reach-visited p*)) ⊆ *marked-as-in True p*
∧ *config-alloc-in-state addr-or-ptr* (*p*, *t*) (*vs*, *s*)
⟹ *config-alloc-in-state addr-or-ptr* (*sw-tr* (*p*, *t*)) (*runST* (*sw-impl-tr vs*) *s*)

If we start our computation with an empty p tree, some of the preconditions of this lemma vanish, as seen by a simple expansion of definitions. A tidier version of our result is then:

theorem *impl-correct-tidied*:
⟦ *t-marked m t*; *t-marked-ext t* {}; *distinct* (*reach t*);
tree-alloc-in-state addr-or-ptr t s;
sw-tr (*Leaf Null*, *t*) = (*p′*, *t′*);
(*runST* (*sw-impl-tr* (*Null*, *addr-or-ptr t*)) *s*) = ((*p-ptr′*, *t-ptr′*), *s′*) ⟧
⟹ *tree-alloc-in-state addr-or-ptr t′ s′* ∧ *t-ptr′* = (*addr-or-ptr t′*)

7 Related Work

Schorr-Waite: At the time of writing of this report, a search on Google Scholar with the keyword "Schorr-Waite" gives about 380 results. Even if some of them are apparently duplicates or only loosely related to the topic, one can assess the interest caused by the original publication [SW67].

Earlier work [BP82,Bir01,Top79,War96] often uses a transformational approach to arrive at an executable program, starting with a high-level relational specification. The first termination proof seems to have been published in [YD77]: Three termination schemas are given, two of which are lexicographic orders (used for simpler variants of the algorithm), one is quite complex, and its well-foundedness proof, even though not obvious, is dismissed as "routine". Our termination argument, substantially different, is a lexicographic combination of three simple measures on trees.

As for mechanization of the proofs, there is the usual divide between interactive theorem proving (that we follow) and fully automated methods that are usually incomplete or cover only very specific correctness properties. On this line, [GM07] carry out a verification by translation of the algorithm to PlusCal and model checking for graphs of bounded size. [LRS06] use the tool TLVA, based on shape analysis. The procedure is not entirely "automatic", as it requires feeding TLVA with appropriate state relations. Even then, the analysis runs for several hours. (By means of comparison, our Isabelle proof script is processed in the order of two minutes.) An advantage of TLVA is that it directly works on C code. It is not quite clear which limitations are effectively imposed on the kind of graph structure (acyclic?) that has been verified.

More recently, there has been some interest in proofs using interactive theorem provers, sparked by Bornat's proof using his Jape prover [Bor00]. This work has later been shortened considerably in Isabelle [MN05], using a "split heap" representation. The proof is directly carried out on an imperative algorithm embedded in the Isabelle system, without any refinement steps. Similar in spirit are proofs using the Caduceus platform [HM05] for a C implementation (also proving termination of the algorithm) and the KeY system [Bub07] for a Java implementation.

Contrasting with this, a proof with the B method [Abr03] follows the refinement tradition of program development, using a total of 8 refinement steps to arrive from an abstract specification of graphs (defined as relation) at an algorithm. In terms of properties shown, this is certainly one of the most complete developments, but some doubts remain: The algorithm is claimed to terminate, but no explicit or implicit statement of finiteness of the graph is made. Note

that in our development, the surjective embedding of inductively defined trees ensures finiteness of the underlying graph.

Representation of memory and imperative programs:

There are two ways to obtain verified executable code: verify written code by abstracting it or generate it from abstract specification.

Haskabelle [RH09] allows to import Haskell code into Isabelle, which can then be used as specification, implementation or intermediate refinement like in [KDE09]. Why/Krakatoa [FM07] is a general framework to generate proof obligations from annotated imperative programs like Java or C into proof assistants like Coq, PVS or Isabelle.

As to the second way, some proof assistants like Coq and Isabelle/HOL are able to generate code from specification in a process called extraction.

The theory Imperative_HOL [BKH+08] takes advantage of Isabelle's extraction and already implements a state-transformer monad with syntax transformations, and code extraction/generation. In this version, the heap is first-order polymorphic, while ours is parameterized, allowing but also needing further instantiations in each case. So one can add any value of one of previously chosen types in our heap, while one can add any value of first-order (or higher but fixed in later versions) type whenever one wants to in the heap of Imperative_HOL.

This polymorphism is obtained by encoding all first-order values as natural numbers. References are also natural numbers with an additional phantom type used to retrieve data from the heap. Indeed, using type reflection, this type is transformed to a value to be given as argument to the heap being a function from natural numbers (addresses) and type representations to natural numbers (values).

There are also other differences:

- We do not have exceptions, our primitives being instead underspecified, for example, in the case of access to null or non-allocated references, because allocation is not managed.
- We use simplification – rewriting – instead of relational reasoning. We have defined rules simplifying execution of primitives with *runST* and *runSR* functions.
- We have a type distinction between state-reader and state-transformer monads allowing to use a nicer syntax and to have static simplifications, as state-readers do not modify the state.

The Ynot project [NMS+08] also uses monads to specify and generate imperative programs within the Coq proof assistant. They have developed a separation logic allowing to simplify reasoning about the heap, while pre- and post-conditions are directly available in the type of computations.

We used a simple memory model as a total function from natural numbers to values which was sufficient in our case, but managing allocation could become hard. Several memory models are compared in [SW09] which then gives a solution combining their advantages which could be interesting.

Reasoning about pointer structures:

The idea to use spanning trees with additional pointers to reason about graph structures is present in the Pale system [MS01], designed to prove the preservation of structural invariants of graphs. Given a graph transformation program, a weakest precondition calculus extracts formulae in monadic second order logic and feeds them to a specialized decision procedure.

Separation logic [ORY01] has recently been advocated as a method of reasoning about pointer programs. The idea is to partition the state into disjoint heap fragments that are manipulated by different code fragments that can be composed modularly. One of the earliest case studies is indeed a proof of the Schorr-Waite algorithm [Yan01]. The proof is not mechanized, but the author has later taken up the example [LYY05] to apply an analysis based on shape grammars. It seems that this analysis only works for trees, and it is not quite clear which properties have effectively been proved.

An interesting recent development is a "tree update logic" [CGZ05,GW09] that combines separation logic with more abstract reasoning about trees: Trees can be decomposed into partial trees with holes, giving rise to separating connectors on a structural level. Details still have to be explored.

8 Conclusions

We have presented a correctness proof of the Schorr-Waite algorithm, starting from a high-level algorithm operating on inductively defined trees (Section 2) to which we add pointers to obtain genuine graphs (Section 5). The low-level algorithm introduced in Section 4 has been proved correct in Section 6 by a simulation argument. The proof in Section 4 for the "pure tree" version of the high-level algorithm has been discussed in more detail for didactic reasons, but is as such immaterial for the full proof presented in Section 6.

The Isabelle proof script is about 1000 lines long and thus compares favorably with previous mechanized proofs of Schorr-Waite, in particular in view of the fact that the termination of the algorithm and structure preservation of the graph after marking have been addressed. It is written in a plain style without particular acrobatics and can presumably be adapted to similar proof assistants without great effort.

Some questions may remain open. The sceptical reader might not be inclined to "believe" some preconditions of our main theorems, for example the precondition *tree-alloc-in-state addr-or-ptr t s* of theorem *impl-correct-tidied* in Section 6. Seen "top-down", the predicate hardly poses a problem: given a tree t with additional pointers, it should not be difficult to lay it out in memory s, by following the recursive procedure embodied by *tree-alloc-in-state*.

The "bottom-up" view is more problematic: given an arbitrary graph in s, can we construct a spanning tree t with *tree-alloc-in-state addr-or-ptr t s* which in addition satisfies *t-marked-ext t* {} (another precondition of theorem *impl-correct-tidied*)? We can argue, informally, that also this view can be accommodated: Construct t by a traversal of the graph structure and accumulate the visited nodes in a set which corresponds to the second argument of *t-marked-ext*

(note that this auxiliary algorithm is just a conceptual device: a constructive existence proof of t and by no means required for the execution of the SW algorithm). A formalization of this sketch could be envisaged, but it would require introducing additional notions (e.g. finiteness of the set of nodes reachable from the starting point of the graph in s) that would hardly contribute to the clarity of the statement.

We think that some essential concepts of our approach (representation of a graph by spanning trees with additional pointers, refinement to imperative programs in monadic style, partitioning of the heap space into subgraphs) can be adapted to other traditional graph algorithms: Often, the underlying structure of interest is indeed tree-shaped, whereas pointers are just used for optimization.

The present paper is primarily a case study, so there are still some rough edges: the representation of our imperative programs has to be refined, with the aim of allowing their compilation to standard programming languages like C or Java. Similarly, we hope to develop patterns that make refinement proofs easier and to partly automate them.

References

[Abr03] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME*, pages 51–74, 2003.

[Bir01] Richard S. Bird. Functional pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11:2001, 2001.

[BKH+08] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, 2008.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic, 2000.

[BP82] Manfred Broy and Peter Pepper. Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. *ACM Trans. Program. Lang. Syst.*, 4:362–381, 1982.

[Bub07] Richard Bubel. The Schorr-Waite-Algorithm. In Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, chapter 15, pages 569–587. Springer Verlag, 2007.

[CGZ05] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Proc. POPL'05*, 2005.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177. Springer, 2007.

[GM07] Miguel Garcia and Ralf Möller. Certification of Transformations Algorithms in Model-driven Software Development. In *Software Engineering 2007*, 2007.

[GW09] Philippa Gardner and Mark Wheelhouse. Small specifications for tree update. In *Proc. WSFM*, 2009.

[HM05] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *In 3rd IEEE Intl. Conf. SEFM'05*, 2005.

[KDE09] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: sel4 — formally verifying a high-performance microkernel. In *Proc. 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2009.

[LRS06] 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.

[LYY05] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *In ESOP*, 2005.

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[RH09] Tobias Rittweiler and Florian Haftmann. Haskabelle – converting Haskell source files to Isabelle/HOL theories, 2009. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/haskabelle.html`.

[SW67] 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.

[SW09] Norbert Schirmer and Makarius Wenzel. State spaces — the locale way. *Electron. Notes Theor. Comput. Sci.*, 254:161–179, 2009.

[Top79] R. W. Topor. The correctness of the Schorr-Waite list marking algorithm. *Acta Informatica*, 11:211–221, 1979.

[War96] Martin Ward. Derivation of data intensive algorithms by formal transformation –the Schorr-Waite graph marking algorithm. *IEEE Transactions on Software Engineering*, 22:665–686, 1996.

[Yan01] Hongseok Yang. An example of local reasoning in bi pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the SPACE Workshop*, 2001.

[YD77] Lawrence Yelowitz and Arthur G. Duncan. Abstractions, instantiations, and proofs of marking algorithms. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 13–21, New York, NY, USA, 1977. ACM.