HAL
archives-ouvertes.fr

Verification of the Schorr-Waite Algorithm - From Trees to Graphs

Mathieu Giorgino, Martin Strecker, Ralph Matthes, Marc Pantel

▶ To cite this version:

Mathieu Giorgino, Martin Strecker, Ralph Matthes, Marc Pantel. Verification of the Schorr-Waite Algorithm - From Trees to Graphs. Logic-Based Program Synthesis and Transformation, Jul 2010, Hagenberg, Austria. pp.67-83, 10.1007/978-3-642-20551-4_5 . hal-00601440

HAL Id: hal-00601440
https://hal.archives-ouvertes.fr/hal-00601440

Submitted on 17 Jun 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verification of the Schorr-Waite algorithm – From trees to graphs

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 [16] 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. Originally conceived to be a particularly space-efficient algorithm to be used in garbage collectors, it has meanwhile become a benchmark for studying pointer algorithms.

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 deterministic 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 graphs with an out-degree of 2 (by a standard encoding of lists by binary trees we can in fact encode 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. In this article, the refinement is carried out manually, but in the long run, we hope to largely automate this step.

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 [12], 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. A longer version of this paper with details of the proofs is available at [7].

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 the ML language. Type variables are written $'a$, $'b$, $\ldots$, 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. $[\![a_0;_{...};a_n]\!] \Longrightarrow b$ abbreviates $a_0 \Longrightarrow (\ldots \Longrightarrow (a_n \Longrightarrow a)\ldots)$. Construction and concatenation operators on lists are represented by $x \# xs$ and xs @ 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* **datatype** $'a$ *tag* = *Tag bool dir* $'a$

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 reversed, 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 (a leaf), and it stops if p is empty again and t is marked. The three operations are illustrated in Figure 1 in which black circles represent marked nodes, white circles unmarked nodes, the directions are indicated by the arrows. Dots indicate intermediate steps and leaves are not represented.

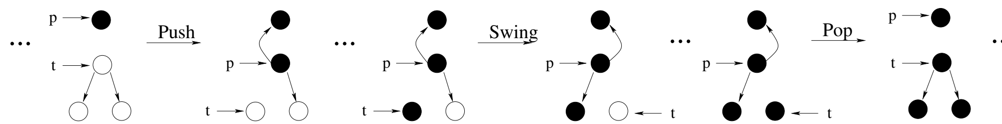

Fig. 1. Operations of the Schorr-Waite algorithm.

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, using the p and t pointers like a zipper data-structure [9]. 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))
```

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 *unmarked-count* and *left-count* with obvious definitions):

```
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])
```

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.

Let's take a look at some invariants of the 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 prematurely, 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:

```
primrec t-marked :: bool ⇒ ('a tag, 'l) tree ⇒ bool where
```

```
  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 that is again a p-shaped tree, and a downwards branch that, depending on the direction, is either a previously marked t tree or an as yet unexplored (and therefore completely unmarked) t tree:

```
primrec p-marked :: ('a tag, 'l) tree ⇒ bool where
  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)))
```

We note in passing that these two properties are invariants of *sw-body*.

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: t-marked True t ⟶ mark-all True R t = t
```

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.

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))
```

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*:
$let\ (p,\ t) = args\ in\ (\forall\ m.\ t\text{-}marked\ m\ t \longrightarrow p\text{-}marked\ p \longrightarrow$
$(let\ (p',\ t') = (sw\text{-}tr\ args)\ in$
$mark\text{-}all\ True\ R\ (reconstruct\ (p,\ t)) = t' \wedge (\exists\ rf.\ p' = Leaf\ rf)))$

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*: $t\text{-}marked\ m\ t \implies sw\text{-}tr\ (Leaf\ rf,\ t) = (p',\ t')$
$\implies t' = mark\text{-}all\ True\ R\ t \wedge (\exists\ rf.\ p' = Leaf\ rf)$

To show the brevity of the development, the full version of the paper [7] reproduces the entire Isabelle script up to this point, which is barely 5 pages long.

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 similar executable code to be generated.

The theory Imperative_HOL [4] 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 code generation for the moment.

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** $('a,\ 's)\ ST = ST\ 's \Rightarrow 'a \times 's$
primrec $runSR :: ('a,\ 's)\ SR \Rightarrow 's \Rightarrow 'a$ **where** $runSR\ (SR\ m) = m$
primrec $runST :: ('a,\ 's)\ ST \Rightarrow 's \Rightarrow 'a \times 's$ **where** $runST\ (ST\ m) = m$
abbreviation *evalST* **where** $evalST\ fm\ s == fst\ (runST\ fm\ s)$

The *return* (also called *unit*) and *bind* functions for manipulating the monads are then defined classically 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** $\unrhd_{SR}$)
  $bindST$ :: $('a,\ 's)\ ST \Rightarrow ('a \Rightarrow ('b,\ 's)\ ST) \Rightarrow ('b,\ 's)\ ST$ (**infixr** $\unrhd_{ST}$)
  $SRtoST$ :: $('a,'s)\ SR \Rightarrow ('a,\ 's)\ ST$

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* ($if\ (a^{SR})\ \{b^{ST}\}\ else\ \{c^{ST}\}$). 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:

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

We finally define the *whileST* combinator ($[v = v0]\ while\ (c)\ \{a\}$), inspired from the *while* combinator definition of the Isabelle/HOL library, the only difference with it being the encapsulation in monads:
$whileST\ b\ c\ v = (doST\{if(\langle b\ v\rangle)\{v' \leftarrow c\ v;\ whileST\ b\ c\ v'\}else\{returnST\ v\}\})$
$while\ b\ c\ v = (if\ b\ v\ then\ while\ b\ c\ (c\ v)\ else\ v)$

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 $('n,\ 'v)\ heap = heap :: 'n \Rightarrow 'v$

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 access-and-update function (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.

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 $('a,\ 'r)\ \mathit{struct} = \mathit{EmptyS} \mid \mathit{Struct}\ 'a\ ('r\ \mathit{ref})\ ('r\ \mathit{ref})$

We define then access-and-update functions $\$v$ (value) $\$l$ (left) and $\$r$ (right) for the $('a,\ 'r)\ \mathit{struct}$ datatype, and access-and-update functions $\$\mathit{mark}$, $\$\mathit{dir}$ and $\$\mathit{val}$ for the $'a\ \mathit{tag}$ datatype.

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 $('r,\ 'a)\ \mathit{str\text{-}heap} = ('r,\ ('a,\ 'r)\ \mathit{struct})\ \mathit{heap}$

An Imperative Algorithm

We now have an idea of the low-level memory representation of trees and can start devising 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) ⇒ (bool, ('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:

constdefs *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}})

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 projections (cf. definition of *tag* in Section 2) :

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 projection 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 t in a state s: Just traverse the tree recursively and check that each node is correctly represented (*tree-alloc-in-state ac t s*). Finally, a configuration (the p and t trees) is correctly represented (*config-alloc-in-state* (p, t)) if each of the trees is, and the p variable contains a reference to the p tree, and similarly for 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:

primrec *addr-of* :: (('r, 'v) *addr tag*, 'b) *tree* ⇒ 'r *ref* **where**
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*)
⟹ *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 *reach* of references occurring in a tree, 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'. We will not go into details here (please refer to the long version of the paper [7]), since the correctness argument for trees is just a light-weight version of the argument for graphs in Section 5.

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*).

For example, the graph of Figure 4 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 2) and a "bad" spanning tree (Figure 3).

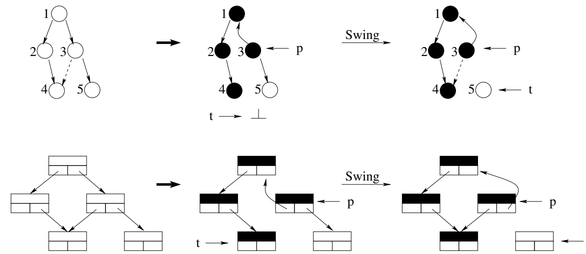

Fig. 2. Low-level graph with a "good" spanning tree

In Figure 2, 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.

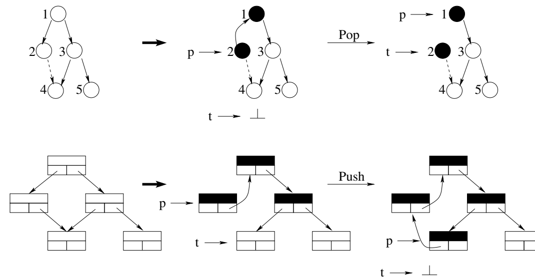

Fig. 3. The same graph with a bad spanning tree

The situation is different in Figure 3, when p reaches node 2. The high-level algorithm finishes its recursive descent at this point and starts backtracking, 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.

primrec *t-marked-ext* :: $(('r,\ 'v)$ *addr tag*, $'r$ *ref*) *tree* $\Rightarrow$ $'r$ *set* $\Rightarrow$ *bool* **where**
  t-marked-ext (*Leaf rf*) *extrefs* = (*case rf of Null* $\Rightarrow$ *True* | *Ref r* $\Rightarrow$ $r \in$ *extrefs*)
| *t-marked-ext* (*Node n l r*) *extrefs* = (*t-marked-ext l* (*insert* (*addr-of-tag n*) *extrefs*)
    $\wedge$ *t-marked-ext r* ((*insert* (*addr-of-tag n*) *extrefs*) $\cup$ *set* (*reach l*)))

There is a similar property *p-marked-ext*, less inspiring, for p trees, which we do not give here. It uses in particular the function *reach-visited* giving the list of addresses of nodes that should have been visited.

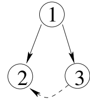

Fig. 4. A tree with additional pointers (drawn as dashed lines)

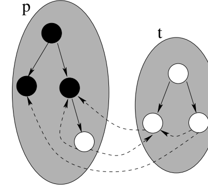

Fig. 5. Partitioning of trees

Let us insist on one 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 5. 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, using the projection *nmark* for retrieving the mark from a tag:

primrec *marked-as-in* :: *bool* $\Rightarrow$ $(('r,\ 'v)$ *addr tag*, $'l$) *tree* $\Rightarrow$ $'r$ *set* **where**
  marked-as-in m (*Leaf rf*) = {}

$|$ *marked-as-in m* (*Node n l r*) = (*if* (*nmark n*) = *m then* {*addr-of-tag n*} *else* {})
$\cup$ *marked-as-in m l* $\cup$ *marked-as-in m r*

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 corresponding 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 parametrize our development with the function *addr-or-ptr* that also returns the references contained in leaf nodes:

primrec *addr-or-ptr* ::(($'r$, $'v$) *addr tag*, $'r$ *ref*) *tree* $\Rightarrow$ $'r$ *ref* **where**
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 establishing the correspondence of the high-level and low-level termination conditions described above.

The proof for the simulation lemma now proceeds essentially along the same lines as before.

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

A considerable amount of work has accumulated on the Schorr-Waite algorithm in particular and on the verification of pointer algorithms in general. For reasons of space, we have to defer a detailed discussion to the full version [7], which also contains some references we cannot accommodate here.

Schorr-Waite: Since its publication [16], the algorithm has given rise to numerous publications which, in the early days, were usually paper proofs which often followed a transformational approach to derive an executable program. There are recently some fully automated methods that, however, are incomplete or cover only very specific correctness properties. Of more interest to us are proofs using interactive theorem provers, sparked by Bornat's proof using his Jape prover [2]. This work has later been shortened considerably in Isabelle [11], using a "split heap" representation. Similar in spirit are proofs using the Caduceus platform [8] and the KeY system [3]. A proof with the B method [1] follows the refinement tradition of program development.

Verification of imperative programs with proof assistants: There are two ways to obtain verified executable code: verify written code by abstracting it or generate it from abstract specification.

Haskabelle [14] allows to import Haskell code into Isabelle, which can then be used as specification, implementation or intermediate refinement like in [10]. Why/Krakatoa [5] 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, the Isabelle extractor generates SML, Caml, Haskell code from executable specifications. The theory Imperative_HOL [4] takes advantage of this and already implements a state-transformer monad with syntax transformations, and code extraction/generation.

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. [15] compares several memory models and then presents a synthesis enjoying their respective advantages, which could be interesting for our work. Our state space partitioning bears similarities with methods advocated in Separation logic [13] – details of this connection 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, to which we add pointers to obtain genuine graphs. The low-level algorithm, written in monadic style, has been proved correct using a refinement relation with the aid of a simulation argument.

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. It favours readability over compactness and can presumably be adapted to similar proof assistants without great effort.

The aim of the present paper is to advocate a development style starting from high level, inductively defined structures and proceeding by refinement. There might be other approaches of verifying low-level heap manipulating algorithms (iterating, for example, over the number of objects stored in the heap and thus not exploiting structural properties). However, we hope to develop patterns that make refinement proofs easier and to partly automate them, thus further decreasing the proof effort.

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. In this spirit, we have verified a BDD construction algorithm [6] and we are planning to apply similar techniques in the context of Model Driven Engineering. Of course, we do not claim that our approach is universally applicable for graphs without deeper structure.

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.

References

1. 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.
2. Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction (MPC)*, LNCS 1837, pages 102–126, 2000.
3. Richard Bubel. The Schorr-Waite-Algorithm. In *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, chapter 15, pages 569–587. Springer Verlag, 2007.
4. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative Functional Programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOL)*, LNCS 5170, 2008.
5. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, LNCS 4590, pages 173–177. Springer, 2007.
6. Mathieu Giorgino and Martin Strecker. Verification of BDD algorithms by refinement of trees. Technical report, IRIT, 2010. `http://www.irit.fr/~Mathieu.Giorgino/Publications/GiSt2010BDD.html`.
7. Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. Verification of the Schorr-Waite algorithm - From trees to graphs, January 2010. `http://www.irit.fr/~Mathieu.Giorgino/Publications/SchorrWaite_TreesGraphs.html`.
8. 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.
9. Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.
10. Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: sel4 — formally verifying a high-performance microkernel. In *International Conference on Functional Programming (ICFP)*. ACM, 2009.
11. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.
12. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.
13. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*, LNCS 2142, pages 1–19. Springer, 2001.
14. 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`.
15. Norbert Schirmer and Makarius Wenzel. State spaces — the locale way. *ENTCS*, 254:161–179, 2009.
16. 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.