Semi-Persistent Data Structures

Sylvain Conchon and Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Orsay, F-91893

Abstract. A data structure is said to be *persistent* when any update operation returns a new structure without altering the old version. This paper introduces a new notion of persistence, called *semi-persistence*, where only ancestors of the most recent version can be accessed or updated. Making a data structure semi-persistent may improve its time and space complexity. This is of particular interest in backtracking algorithms manipulating persistent data structures, where this property is usually satisfied. We propose a proof system to statically check the valid use of semi-persistent data structures. It requires a few annotations from the user and then generates proof obligations that are automatically discharged by a dedicated decision procedure.

1 Introduction

A data structure is said to be *persistent* when any update operation returns a new structure without altering the old version. In purely applicative programming, data structures are automatically persistent [16]. Yet this notion is more general and the exact meaning of persistent is *observationally immutable*. Driscoll et al. even proposed systematic techniques to make imperative data structures persistent [9]. In particular, they distinguish *partial* persistence, where all versions can be accessed but only the newest can be updated, from *full* persistence where any version can be accessed or updated. In this paper, we study another notion of persistence, which we call *semi-persistence*.

One of the main interests of a persistent data structure shows up when it is used within a *backtracking* algorithm. Indeed, when we are back from a branch, there is no need to undo the modifications performed on the data structure: we simply use the old version, which persisted, and start a new branch. One can immediately notice that full persistence is not needed in this case, since we are reusing *ancestors* of the current version, but never *siblings* (in the sense of another version obtained from a common ancestor). We shall call *semi-persistent* a data structure where only ancestors of the newest version can be updated. Note that this notion is different from partial persistence, since we need to update ancestors, and not only to access them.

A semi-persistent data structure can be more efficient than its fully persistent counterpart, both in time and space. An algorithm using a semi-persistent data structure may be written as if it was operating on a fully persistent data structure, *provided that we only backtrack to ancestor versions*. Checking the

correctness of a program involving a semi-persistent data structure amounts to showing that

- first, the data structure is *correctly used*;
- second, the data structure is *correctly implemented*.

This article only addresses the former point. Regarding the latter, we simply give examples of semi-persistent data structures. Proving the correctness of these implementations is out of the scope of this paper (see Section 5).

Our approach consists in annotating programs with user pre- and postconditions, which mainly amounts to expressing the validity of the successive versions of a semi-persistent data structure. By validity, we mean being an ancestor of the newest version. Then we compute a set of proof obligations which express the correctness of programs using a weakest precondition-like calculus [8]. These obligations lie in a decidable logical fragment, for which we provide a sound and complete decision procedure. Thus we end up with an almost automatic way of checking the legal use of semi-persistent data structures.

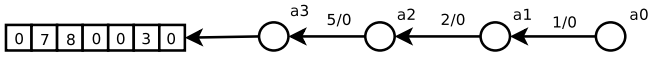
Related work. To our knowledge, this notion of semi-persistence is new. However, there are several domains which are somehow connected to our work, either because they are related to some kind of stack analysis, or because they are providing decision procedure for reachability issues. First, works on escape analysis [12, 4] address the problem of stack-allocating values; we may think that semi-persistent versions that become invalid are precisely those which could be stack-allocated, but it is not the case as illustrated by example g above. Second, works on stack analysis to ensure memory safety [14, 18, 19] provide methods to check the consistent use of push and pop operations. However, these approaches are not precise enough to distinguish between two sibling versions (of a given semi-persistent data structure) which are both upper in the stack. Regarding the decidability of our proof obligations, our approach is similar to other works regarding reachability in linked data structures [15, 3, 17]. However, our logic is much simpler and we provide a specific decision procedure. Finally, we can mention Knuth's *dancing links* [13] as an example of a data structure specifically designed for backtracking algorithms; but it is still a traditional imperative solution where an explicit undo operation is performed in the main algorithm.

This paper is organized as follows. First, Section 2 gives examples of semi-persistent data structures and shows the benefits of semi-persistence with some benchmarks. Then our formalization of semi-persistence is presented in two steps: Section 3 introduces a small programming language to manipulate semi-persistent data structures, and Section 4 defines the proof system which checks the valid use of semi-persistent data structures. Section 5 concludes with possible extensions. A long version of this paper, including proofs, is available online [7].

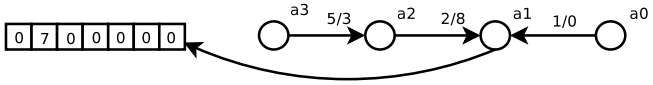
2 Examples of Semi-Persistent Data Structures

We explain how to implement semi-persistent arrays, lists and hash tables and we present benchmarks to show the benefits of semi-persistence.

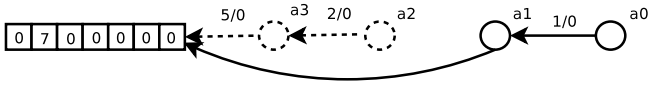
Arrays. Semi-persistent arrays can be implemented by modifying the persistent arrays introduced by Baker [1]. The basic idea is to use an imperative array for the newest version of the persistent array and indirections for old versions. For instance, starting with an array a_0 initialized with 0, and performing the successive updates $a_1 = \texttt{set}(a_0, 1, 7)$, $a_2 = \texttt{set}(a_1, 2, 8)$ and $a_3 = \texttt{set}(a_2, 5, 3)$, we end up with the following situation:



When accessing or updating an old version, e.g. a_1, Baker's solution is to first perform a *rerooting* operation, which makes a_1 point to the imperative array by reversing the linked list of indirections:



But if we know that we are not going to access a_2 and a_3 anymore, we can save this list reversal. All we need to do is to perform the assignments contained in this list:



Thus it is really easy to turn these persistent arrays into a semi-persistent data structure, which is more efficient since we save some pointer assignments. This example is investigated in more details in [6].

Lists. As a second example, we consider an immutable data structure which we make semi-persistent. The simplest and most popular example is the list data structure. To make it semi-persistent, the idea is to reuse *cons* cells between successive conses to the same list. For instance, given a list `l`, the cons operation `1::l` allocates a new memory block to store `1` and a pointer to `l`. Then a successive operation `2::l` could reuse the same memory block if the list is used in a semi-persistent way. Thus we simply need to replace `1` by `2`. To do this, we must maintain for each list the previous cons, if any.

Hash Tables. Combining (semi-)persistent arrays with (semi-)persistent lists, one easily gets (semi-)persistent hash tables.

Benchmarks. We present some benchmarks to show the benefits of semi-persistence. Each of the previous three data structures has been implemented in Ocaml[1]. Each data structure is tested the same way and compared to its fully persistent counterpart. The test consists in simulating a backtracking algorithm with branching degree 4 and depth 6, operating on a single data structure. N

[1] The full code is available in the long version of this paper [7].

successive update operations are performed on the data structure between two branchings points.

The following table gives timings for various values of N. The code was compiled with the Ocaml native-code compiler (`ocamlopt -unsafe`) on a dual core Pentium 2.13GHz processor running under Linux. The timings are given in seconds and correspond to CPU time obtained using the UNIX `times` system call.

N	200	1000	5000	10000
persistent arrays	0.21	1.50	13.90	30.5
semi-persistent arrays	0.18	1.10	7.59	17.3
persistent lists	0.18	2.38	50.20	195.0
semi-persistent lists	0.11	0.76	8.02	31.1
persistent hash tables	0.24	2.15	19.30	43.1
semi-persistent hash tables	0.22	1.51	11.20	28.2

As we can see, the speedup ratio is always greater than 1 and almost reaches 7 (for semi-persistent lists). Regarding memory consumption, we compared the total number of allocated bytes, as reported by Ocaml's garbage collector. For the tests corresponding to the last column ($N = 10000$) semi-persistent data structures always used much less memory than persistent ones: 3 times less for arrays, 575 times less for lists and 1.5 times less for hash tables. The dramatic ratio for lists is easily explained by the fact that our benchmark program reflects the best case regarding memory allocation (allocation in one branch is reused in other branches, which all have the same length).

3 Programming with Semi-Persistent Data Structures

This section introduces a small programming language to manipulate semi-persistent data structures. In order to keep it simple, we assume that we are operating on the successive versions of a single, statically allocated, data structure. Multiple data structures and dynamic allocation are discussed in Section 5.

3.1 Syntax

The syntax of our language is as follows:

$$
\begin{array}{l}
e ::= x \mid c \mid p \mid f\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
\qquad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
d ::= \texttt{fun}\ f\ (x : \iota) = \{\phi\}\, e\, \{\psi\} \\
\iota ::= \texttt{semi} \mid \delta \mid \texttt{bool}
\end{array}
$$

A program expression is either a variable (x), a constant (c), a pointer (p), a function call, a local variable introduced by a `let` binding, or a conditional. The set of function names f includes some primitive operations (introduced in

the next section). A function definition d introduces a function f with exactly one argument x of type ι, a precondition ϕ, a body and a postcondition ψ. A type ι is either the type `semi` of the semi-persistent data structure, the type δ of the values it contains, or the type `bool` of booleans. The syntax of pre- and postconditions will be given later in Section 4. A program Δ is a finite set of mutually recursive functions.

3.2 Primitive Operations

We may consider three kinds of operations on semi-persistent data structures: *update* operations backtracking to a given version and creating a new successor, which becomes the newest version; *destructive access* operations backtracking to a given version, which becomes the newest version, and then accessing it; and *non-destructive access* operations accessing an ancestor of the newest version, without modifying the data structure.

Since update and destructive access operations both need to backtrack, it is convenient to design a language based on the following three primitives: `backtrack`, which backtracks to a given version, making it the newest version; `branch` which builds a new successor of a given version, assuming it is the newest version; and `acc`, which accesses a given version, assuming it is a valid version. Then update and destructive access operations can be rephrased in terms of the above primitives:

$$\begin{aligned} \texttt{upd}\ e &= \texttt{branch}\ (\texttt{backtrack}\ e) \\ \texttt{dacc}\ e &= \texttt{acc}\ (\texttt{backtrack}\ e) \end{aligned}$$

3.3 Operational Semantics

We equip our language with a small step operational semantics, which is given in Figure 1. One step of reduction is written $e_1, S_1 \to e_2, S_2$ where e_1 and e_2 are program expressions and S_1 and S_2 are states. A value v is either a constant c or a pointer p. Pointers represent versions of the semi-persistent data structure. A state S is a stack $p_1, \ldots, p_m$ of pointers, p_m being the top of the stack. The semantics is straightforward, except for primitive operations. Primitive `backtrack` expects an argument p_n designating a valid version of the data structure, that is an element of the stack. Then all pointers on top of p_n are popped from the stack and p_n is the result of the operation. Primitive `branch` expects an argument p_n being the top of the stack and pushes a new value p, which is also the result of the operation. Finally, primitive `acc` expects an argument p_n designating a valid version, leaves the stack unchanged and returns some value for version p_n, represented by $\mathcal{A}(p_n)$. (We leave $\mathcal{A}$ uninterpreted since we are not interested in the values contained in the data structure.)

Note that reduction of `backtrack` p_n or `acc` p_n is blocked whenever p_n is not an element of S, which is precisely what we intend to prevent.

$$
\begin{array}{rcl}
E & ::= & [] \mid f\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
v & ::= & c \mid p \\
S & ::= & p \cdots p
\end{array}
$$

$$
\begin{array}{rcll}
\texttt{if true then}\ e_1\ \texttt{else}\ e_2, S & \rightarrow & e_1, S & \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2, S & \rightarrow & e_2, S & \\
\texttt{let}\ x = v\ \texttt{in}\ e, S & \rightarrow & e\{x \leftarrow v\}, S & \\
f\ v, S & \rightarrow & e\{x \leftarrow v\}, S & \text{if } \texttt{fun}\, f\ (x : \iota) = \{\phi\}\, e\, \{\psi\} \in \Delta \\
\texttt{backtrack}\ p_n, p_1 \cdots p_n p_{n+1} \cdots p_m & \rightarrow & p_n, p_1 \cdots p_n & \\
\texttt{branch}\ p_n, p_1 \cdots p_n & \rightarrow & p, p_1 \cdots p_n p & p \text{ fresh} \\
\texttt{acc}\ p_n, p_1 \cdots p_n p_{n+1} \cdots p_m & \rightarrow & \mathcal{A}(p_n), p_1 \cdots p_n p_{n+1} \cdots p_m & \\
E[e_1], S_1 & \rightarrow & E[e_2], S_2 & \text{if } e_1, S_1 \rightarrow e_2, S_2 \text{ and } E \neq []
\end{array}
$$

Fig. 1. Operational Semantics

3.4 Type System with Effect

We introduce a type system to characterize well-formed programs. Our language is simply typed and thus type-checking is immediate. Meanwhile, we infer the effect ϵ of each expression, as an element of the boolean lattice $(\{\bot, \top\}, \wedge, \vee)$. This boolean indicates whether the expression modifies the semi-persistent data structure ($\bot$ meaning no modification and $\top$ a modification). Effects will be used in the next section to simplify constraint generation. Each function is given a type τ, as follows:

$$\tau ::= (x : \iota) \rightarrow^{\epsilon} \{\phi\}\, \iota\, \{\psi\}$$

The argument is given a type and a name (x) since it is bound in both precondition ϕ and postcondition ψ. Type τ also indicates the latent effect ϵ of the function, which is the effect resulting from the function application.

A typing environment Γ is a set of type assignments for variables $(x : \iota)$, constants $(c : \iota)$ and functions $(f : \tau)$. It is assumed to contain at least type declarations for the primitives, as follows:

$$
\begin{array}{rcl}
\texttt{backtrack} & : & (x : \texttt{semi}) \rightarrow^{\top} \{\phi_{\texttt{backtrack}}\}\, \texttt{semi}\, \{\psi_{\texttt{backtrack}}\} \\
\texttt{branch} & : & (x : \texttt{semi}) \rightarrow^{\top} \{\phi_{\texttt{branch}}\}\, \texttt{semi}\, \{\psi_{\texttt{branch}}\} \\
\texttt{acc} & : & (x : \texttt{semi}) \rightarrow^{\bot} \{\phi_{\texttt{acc}}\}\, \delta\, \{\psi_{\texttt{acc}}\}
\end{array}
$$

where pre- and postcondition are given later. As expected, both `backtrack` and `branch` modify the semi-persistent data structure and thus have effect $\top$, while the non-destructive access `acc` has effect $\bot$.

Given a typing environment Γ, the judgment $\Gamma \vdash e : \iota, \epsilon$ means "e is a well-formed expression of type ι and effect ϵ" and the judgment $\Gamma \vdash d : \tau$ means "d is a well-formed function definition of type τ". Typing rules are given in Figure 2. They assume judgments $\Gamma \vdash \phi$ pre and $\Gamma \vdash \psi$ post ι for the well-formedness of pre- and postconditions respectively, to be defined later in Section 4.1. Note that there is no typing rule for pointers, to prevent their explicit use in programs.

$$\textsc{Var}\ \frac{x : \iota \in \Gamma}{\Gamma \vdash x : \iota, \bot} \qquad \textsc{Const}\ \frac{c : \iota \in \Gamma}{\Gamma \vdash c : \iota, \bot}$$

$$\textsc{App}\ \frac{f : (x : \iota_1) \to^{\epsilon_2} \{\phi\}\, \iota_2\, \{\psi\} \in \Gamma \qquad \Gamma \vdash e : \iota_1, \epsilon_1}{\Gamma \vdash f\ e : \iota_2, \epsilon_1 \vee \epsilon_2}$$

$$\textsc{Ite}\ \frac{\Gamma \vdash e_1 : \texttt{bool}, \epsilon_1 \qquad \Gamma \vdash e_2 : \iota, \epsilon_2 \qquad \Gamma \vdash e_3 : \iota, \epsilon_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \iota, \epsilon_1 \vee \epsilon_2 \vee \epsilon_3}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 : \iota_1, \epsilon_1 \qquad \Gamma, x : \iota_1 \vdash e_2 : \iota_2, \epsilon_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \iota_2, \epsilon_1 \vee \epsilon_2}$$

$$\textsc{Fun}\ \frac{x : \iota_1 \vdash \phi\ \mathsf{pre} \qquad x : \iota_1 \vdash \psi\ \mathsf{post}\ \iota_2 \qquad \Gamma, x : \iota_1 \vdash e : \iota_2, \epsilon}{\Gamma \vdash \texttt{fun}\, f\ (x : \iota_1) = \{\phi\}\, e\, \{\psi\} : (x : \iota_1) \to^{\epsilon} \{\phi\}\, \iota_2\, \{\psi\}}$$

Fig. 2. Typing Rules

A program $\Delta = d_1, \ldots, d_n$ is well-typed if each function definition d_i can be given a type τ_i such that $d_1 : \tau_1, \ldots, d_n : \tau_n \vdash d_i : \tau_i$ for each i. The types τ_i can easily be obtained by a fixpoint computation, starting will all latent effects set to $\bot$, since effect inference is clearly a monotone function.

3.5 Examples

Let us consider the following two functions f and g:

$\texttt{fun}\ f\ x_0 = \{\texttt{valid}(x_0)\}$	$\texttt{fun}\ g\ x_0 = \{\texttt{valid}(x_0)\}$
$\quad\texttt{let}\ x_1 = \texttt{upd}\ x_0\ \texttt{in}$	$\quad\texttt{let}\ x_1 = \texttt{upd}\ x_0\ \texttt{in}$
$\quad\texttt{let}\ x_2 = \texttt{upd}\ x_0\ \texttt{in}$	$\quad\texttt{let}\ x_2 = \texttt{upd}\ x_0\ \texttt{in}$
$\quad\texttt{acc}\ x_2$	$\quad\texttt{acc}\ x_1$

Each function expects a valid version x_0 of the data structure as argument and successively build two successors x_1 and x_2 of x_0. Then f accesses x_2, which is valid, and g accesses x_1, which is illegal. Let us check this on the operational semantics. Let S be a state composed of a single pointer p. The reduction of $f\ p$ in S runs as follows:

$$\begin{aligned} f\ p, p &\to \texttt{let}\ x_1 = \texttt{upd}\ p\ \texttt{in}\ \texttt{let}\ x_2 = \texttt{upd}\ p\ \texttt{in}\ \texttt{acc}\ x_2, p \\ &\to \texttt{let}\ x_1 = p_1\ \texttt{in}\ \texttt{let}\ x_2 = \texttt{upd}\ p\ \texttt{in}\ \texttt{acc}\ x_2, pp_1 \\ &\to \texttt{let}\ x_2 = \texttt{upd}\ p\ \texttt{in}\ \texttt{acc}\ x_2, pp_1 \\ &\to \texttt{let}\ x_2 = p_2\ \texttt{in}\ \texttt{acc}\ x_2, pp_2 \\ &\to \texttt{acc}\ p_2, pp_2 \\ &\to \mathcal{A}(p_2), pp_2p_3 \end{aligned}$$

and ends on the value $\mathcal{A}(p_2)$. On the contrary, the reduction of $g\ p$ in S blocks on $g\ p, p \to \ldots \to \texttt{acc}\ p_1, pp_2$.

4 Proof System

This section introduces a theory for semi-persistence and a proof system for this theory. First we define the syntax and semantics of logical annotations. Then we compute a set of constraints for each program expression, which is proved to express the correctness of the program with respect to semi-persistence. Finally we give a decision procedure to solve the constraints.

4.1 Theory of Semi-Persistence

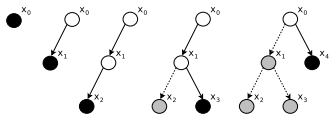
The syntax of annotations is as follows:

$$\begin{aligned} \text{term } t &::= x \mid p \mid \texttt{prev}(t) \\ \text{atom } a &::= t = t \mid \texttt{path}(t, t) \\ \text{postcondition } \psi &::= a \mid \psi \wedge \psi \\ \text{precondition } \phi &::= a \mid \phi \wedge \phi \mid \psi \Rightarrow \phi \mid \forall x.\, \phi \end{aligned}$$

Terms are built from variables, pointers and a single function symbol `prev`. Atoms are built from equality and a single predicate symbol `path`. A postcondition ψ is restricted to a conjunction of atoms. A precondition is a formula ϕ built from atoms, conjunctions, implications and universal quantifications. A negative formula (i.e. appearing on the left side of an implication) is restricted to a conjunction of atoms. We introduce two different syntactic categories ψ and ϕ for formulae but one can notice that ϕ actually contains ψ. This syntactic restriction on formulae is justified later in Section 4.5 when introducing the decision procedure. In the remainder of the paper, a "formula" refers to the syntactic category ϕ. Substitution a of term t for a variable x in a formula ϕ is written $\phi\{x \leftarrow t\}$. We denote by $\mathcal{S}(A)$ the set of all subterms of a set of atoms A.

The typing of terms and formulae is straightforward, assuming that `prev` has signature `semi` $\rightarrow$ `semi`. Function postconditions may refer to the function result, represented by the variable *ret*. Formulae can only refer to variables of type `semi` (including variable *ret*). We write $\Gamma \vdash \phi$ to denote a well-formed formula ϕ in a typing environment Γ.

We now give the semantics of program annotations. The main idea is to express that a given version is valid if and only if it is an ancestor of the newest version. To illustrate this idea, the following figure shows the successive version trees for the sequence of declarations $x_1 =$ `upd` x_0, $x_2 =$ `upd` x_1, $x_3 =$ `upd` x_1 and $x_4 =$ `upd` x_0:



The newest version is pictured as a black node, other valid versions as white nodes and invalid ones as gray nodes.

The meaning of `prev` and `path` is to define the notion of ancestor: `prev`(x) is the immediate ancestor of x and `path`(x, y) holds whenever x is an ancestor of y. The corresponding theory can be axiomatized as follows:

Definition 1. *The theory $\mathcal{T}$ is defined as the combination of the theory of equality and the following axioms:*

$$(A_1)\ \forall x.\, \mathtt{path}(x, x)$$
$$(A_2)\ \forall xy.\, \mathtt{path}(x, \mathtt{prev}(y)) \Rightarrow \mathtt{path}(x, y)$$
$$(A_3)\ \forall xyz.\, \mathtt{path}(x, y) \wedge \mathtt{path}(y, z) \Rightarrow \mathtt{path}(x, z)$$

We write $\models \phi$ if ϕ is valid in any model of $\mathcal{T}$.

The three axioms (A_1)–(A_3) exactly define `path` as the reflexive transitive closure of $\mathtt{prev}^{-1}$, since we consider validity in all models of $\mathcal{T}$ and therefore in those where `path` is the smallest relation satisfying axioms (A_1)–(A_3). Note that `prev` is a total function and that there is no notion of "root" in our logic. Thus a version always has an immediate ancestor, which may or may not be valid.

To account for the modification of the newest version as program execution progresses, we introduce a "mutable" variable *cur* to represent the newest version. This variable does not appear in programs: its scope is limited to annotations. The only way to modify its contents is to call the primitive operations `backtrack` and `branch`. We are now able to give the full type expressions for the three primitive operations:

$$\begin{array}{l}
\mathtt{backtrack}: \\
\quad (x : \mathtt{semi}) \to^{\top} \{\mathtt{path}(x, \mathit{cur})\}\, \mathtt{semi}\, \{\mathit{ret} = x \wedge \mathit{cur} = x\} \\
\mathtt{branch}: \\
\quad (x : \mathtt{semi}) \to^{\top} \\
\qquad \{\mathit{cur} = x\}\, \mathtt{semi}\, \{\mathit{ret} = \mathit{cur} \wedge \mathtt{prev}(\mathit{cur}) = x\} \\
\mathtt{acc}: \\
\quad (x : \mathtt{semi}) \to^{\bot} \{\mathtt{path}(x, \mathit{cur})\}\, \delta\, \{\mathtt{true}\}
\end{array}$$

As expected, effect $\top$ for the first two reflects the modification of *cur*. The validity of function argument x is expressed as $\mathtt{path}(x, \mathit{cur})$ in operations `backtrack` and `acc`. Note that `acc` has no postcondition (written `true` and which could stand for the tautology $\mathit{cur} = \mathit{cur}$) since we are not interested in the values contained in the data structure.

We are now able to define the judgements used in Section 3.4 for pre- and postconditions. We write $\Gamma \vdash \phi$ pre as syntactic sugar for $\Gamma, \mathit{cur} : \mathtt{semi} \vdash \phi$. Similarly, $\Gamma \vdash \psi$ post ι is syntactic sugar for $\Gamma, \mathit{cur} : \mathtt{semi}, \mathit{ret} : \iota \vdash \psi$ when return type ι is `semi` and for $\Gamma, \mathit{cur} : \mathtt{semi} \vdash \psi$ otherwise. Note that since Γ only contains the function argument x in typing rule Fun, the function precondition may only refer to x and *cur*, and its postcondition to x, *cur* and *ret*.

4.2 Constraints

We now define the set of constraints associated to a given program. This is mostly a weakest precondition calculus, which is greatly simplified here since we have only one mutable variable (namely *cur*). For a program expression e and a formula ϕ we write this weakest precondition $\mathcal{C}(e, \phi)$. This is formula expressing

$$\begin{aligned}
\texttt{frame}_f(\phi) &= \phi_f\{x \leftarrow ret\} \wedge \forall ret'.\, \psi_f\{ret \leftarrow ret', x \leftarrow ret\} \Rightarrow \phi\{ret \leftarrow ret'\} \\
&\quad \text{if } f : (x : \iota) \rightarrow^{\bot} \{\phi_f\}\, \iota'\, \{\psi_f\} \\
\texttt{frame}_f(\phi) &= \phi_f\{x \leftarrow ret\} \wedge \forall ret'\, cur'.\, \psi_f\{ret \leftarrow ret', x \leftarrow ret, cur \leftarrow cur'\} \Rightarrow \\
&\qquad\qquad \phi\{ret \leftarrow ret', cur \leftarrow cur'\} \\
&\quad \text{if } f : (x : \iota) \rightarrow^{\top} \{\phi_f\}\, \iota'\, \{\psi_f\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}(v, \phi) &= \phi\{ret \leftarrow v\} \\
\mathcal{C}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \phi) &= \mathcal{C}(e_1, \mathcal{C}(e_2, \phi) \wedge \mathcal{C}(e_3, \phi)) \\
\mathcal{C}(\texttt{let } x = e_1 \texttt{ in } e_2, \phi) &= \mathcal{C}(e_1, \mathcal{C}(e_2, \phi)\{x \leftarrow ret\}) \\
\mathcal{C}(f\ e_1, \phi) &= \mathcal{C}(e_1, \texttt{frame}_f(\phi)) \\
\mathcal{C}(\texttt{fun } f\ (x : \iota) = \{\phi\}\, e\, \{\psi\}) &= \forall x.\, \forall cur.\, \phi \Rightarrow \mathcal{C}(e, \psi)
\end{aligned}$$

Fig. 3. Constraint synthesis

the conditions under which ϕ will hold after the evaluation of e. Note that *cur* may appear in ϕ, denoting the result of e, but does not appear in $\mathcal{C}(e, \phi)$ anymore. For a function definition d we write $\mathcal{C}(d)$ the formula expressing its correctness, that is the fact that the function precondition implies the weakest precondition obtained from the function postcondition, for any function argument and any initial value of *cur*. The definition for $\mathcal{C}(e, \phi)$ is given in Figure 3. This is a standard weakest precondition calculus, except for the conditional rule. Indeed, one would expect a rule such as

$$\begin{aligned}
&\mathcal{C}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \phi) = \\
&\ \mathcal{C}(e_1, (ret = \texttt{true} \Rightarrow \mathcal{C}(e_2, \phi)) \wedge (ret = \texttt{false} \Rightarrow \mathcal{C}(e_3, \phi)))
\end{aligned}$$

but since ϕ cannot test the result of condition e_1 (ϕ may only refer to variables of type `semi`), the conjunction above simplifies to $\mathcal{C}(e_2, \phi) \wedge \mathcal{C}(e_3, \phi)$.

The constraint synthesis for a function call, $\mathcal{C}(f\ e_1, \phi)$, is the only nontrivial case. It requires precondition ϕ_f to be valid and postcondition ψ_f to imply the expected property ϕ. Universal quantification is used to introduce f's results and side-effects. We use the effect in f's type to distinguish two cases: either effect is $\bot$ which means that *cur* is not modified and thus we only quantify over f's result (hence we get for free the invariance of *cur*); or effect is $\top$ and we quantify over an additional variable cur' which stands for the new value of *cur*. To simplify this definition, we introduce a formula transformer $\texttt{frame}_f(\phi)$ which builds the appropriate postcondition for argument e_1.

4.3 Examples

Simple Example. Let us consider again the two functions f and g from Section 3.5, $\texttt{valid}(x_0)$ being now expressed as $\texttt{path}(x_0, cur)$. We compute the as-

sociated constraints for an empty postcondition $\texttt{true}$. The constraint $\mathcal{C}(f)$ is

$$\begin{array}{l}\forall x_0.\,\forall cur.\,\texttt{path}(x_0, cur) \Rightarrow \\ \quad \texttt{path}(x_0, cur) \wedge \\ \quad \forall x_1.\,\forall cur_1.\,(\texttt{prev}(x_1) = x_0 \wedge cur_1 = x_1) \Rightarrow \\ \quad\quad \texttt{path}(x_0, cur_1) \wedge \\ \quad\quad \forall x_2.\,\forall cur_2.\,(\texttt{prev}(x_2) = x_0 \wedge cur_2 = x_2) \Rightarrow \\ \quad\quad\quad \texttt{path}(x_2, cur_2) \ \wedge \forall ret.\,\texttt{true} \Rightarrow \texttt{true}\end{array}$$

It can be split into three proof obligations, which are the following universally quantified sequents:

$$\texttt{path}(x_0, cur) \vdash \texttt{path}(x_0, cur)$$

$$\texttt{path}(x_0, cur), \texttt{prev}(x_1) = x_0,\, cur_1 = x_1 \vdash \texttt{path}(x_0, cur_1)$$

$$\begin{array}{c}\texttt{path}(x_0, cur), \texttt{prev}(x_1) = x_0, \\ cur_1 = x_1, \texttt{prev}(x_2) = x_0,\, cur_2 = x_2 \vdash \texttt{path}(x_2, cur_2)\end{array}$$

The three of them hold in theory $\mathcal{T}$ and thus f is correct. Similarly, the constraint $\mathcal{C}(g)$ can be computed and split into three proof obligations. The first two are exactly the same as for f but the third one is slightly different:

$$\begin{array}{c}\texttt{path}(x_0, cur), \texttt{prev}(x_1) = x_0, \\ cur_1 = x_1, \texttt{prev}(x_2) = x_0,\, cur_2 = x_2 \vdash \texttt{path}(x_1, cur_2)\end{array}$$

In that case it does not hold in theory $\mathcal{T}$.

Backtracking Example. As a more complex example, let us consider a backtracking algorithm. The pattern of a program performing backtracking on a persistent data structure is a recursive function bt looking like

$$\texttt{fun}\ bt\ (x : \texttt{semi})\ = \ldots\ bt(\texttt{upd}\ x)\ \ldots\ bt(\texttt{upd}\ x)\ \ldots$$

Function bt takes a data structure x as argument and makes recursive calls on several successors of x. This is precisely a case where the data structure may be semi-persistent, as motivated in the introduction. To capture this pattern in our framework, we simply need to consider two successive calls $bt(\texttt{upd}\ x)$, which can be written as follows:

$$\begin{array}{l}\texttt{fun}\ bt\ (x : \texttt{semi})\ = \\ \quad \texttt{let}\ \_ = bt(\texttt{upd}\ x)\ \texttt{in}\ bt(\texttt{upd}\ x)\end{array}$$

Function bt obviously requires a precondition stating that x is a valid version of the semi-persistent data structure. This is not enough information to discharge the proof obligations: the second recursive call $bt(\texttt{upd}\ x)$ requires x to be valid, which possibly could no longer be the case after the first recursive call. Therefore a postcondition for bt is needed to ensure the validity of x:

$$\begin{array}{l}\texttt{fun}\ bt\ (x : \texttt{semi})\ = \\ \quad \{\ \texttt{path}(x, cur)\ \} \\ \quad\quad \texttt{let}\ \_ = bt(\texttt{upd}\ x)\ \texttt{in}\ bt(\texttt{upd}\ x) \\ \quad \{\ \texttt{path}(x, cur)\ \}\end{array}$$

Then it is straightforward to check that constraint $\mathcal{C}(bt)$ is valid in theory $\mathcal{T}$.

4.4 Soundness

In the remainder of this section, we consider a program $\Delta = d_1, \ldots, d_n$ whose constraints are valid, that is $\models \mathcal{C}(d_1) \wedge \cdots \wedge \mathcal{C}(d_n)$. We are going to show that the evaluation of this program will not block.

For this purpose we first introduce the notion of validity with respect to a state of the operational semantics:

Definition 2. *A formula ϕ is valid in a state $S = p_1, \ldots, p_n$, written $S \models \phi$, if it is valid in any model $\mathcal{M}$ for $\mathcal{T}$ such that*

$$\begin{cases} \mathtt{prev}(p_{i+1}) = p_i \quad \textit{for all } 1 \leq i < n \\ \mathit{cur} = p_n \end{cases}$$

Then we show that this validity is preserved by the operational semantics. To do this, it is convenient to see the evaluation contexts as formula transformers, as follows:

E	$E[\phi]$
$[]$	ϕ
$\mathtt{let}\ x = E_1\ \mathtt{in}\ e_2$	$E_1[\mathcal{C}(e_2, \phi)\{x \leftarrow \mathit{ret}\}]$
$\mathtt{if}\ E_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$	$E_1[\mathcal{C}(e_2, \phi) \wedge \mathcal{C}(e_3, \phi)]$
$f\ E_1$	$E_1[\mathtt{frame}_f(\phi)]$

There is a property of commutation between contexts for programs and contexts for formulae:

Lemma 1. *$S \models \mathcal{C}(E[e], \phi)$ if and only if $S \models \mathcal{C}(e, E[\phi])$.*

We now want to prove preservation of validity, that is if $S \models \mathcal{C}(e, \phi)$ and $e, S \rightarrow e', S'$ then $S' \models \mathcal{C}(e', \phi)$. Obviously, this does not hold for any state S, program e and formula ϕ. Indeed, if $S \equiv p_1 p_2$, $e \equiv \mathtt{upd}\ p_1$ and $\phi \equiv \mathtt{prev}(p_2) = p_1$, then $\mathcal{C}(e, \phi)$ is

$$\mathtt{path}(p_1, \mathit{cur}) \ \wedge \forall \mathit{ret}'\, \mathit{cur}'.\, (\mathtt{prev}(\mathit{ret}') = p_1 \wedge \mathit{cur}' = \mathit{ret}') \Rightarrow \mathtt{prev}(p_2) = p_1$$

which holds in S. But $S' \equiv p_1 p$ for a fresh p, $e' \equiv p$, and $\mathcal{C}(e', \phi)$ is $\mathtt{prev}(p_2) = p_1$ which does not hold in S' (since p_2 does not appear in S' anymore). Fortunately, we are not interested in the preservation of $\mathcal{C}(e, \phi)$ for any formula ϕ, but only for formulae which arise from function postconditions. As pointed out in Section 4.1, a function postcondition may only refer to x, *cur* and *ret* only. Therefore we are only considering formulae $\mathcal{C}(e, \phi)$ where x is the only free variable (*cur* and *ret* do not appear in formulae $\mathcal{C}(e, \phi)$ anymore). This excludes the formula $\mathtt{prev}(p_2) = p_1$ in the example above.

We are now able to prove preservation of validity:

Lemma 2. *Let S be a state, ϕ be a formula and e a program expression. If $S \models \mathcal{C}(e, \phi)$ and $e, S \rightarrow e', S'$ then $S' \models \mathcal{C}(e', \phi)$.*

Finally, we prove the following progress property:

Theorem 1. *Let S be a state, ϕ be a formula and e a program expression. If $S \models \mathcal{C}(e, \phi)$ and $e, S \rightarrow^* e', S' \not\rightarrow$, then e' is a value.*

4.5 Decision Procedure

We now show that constraints are decidable and we give a decision procedure. First, we notice that any formula ϕ is equivalent to a conjunction of formulae of the form $\forall x_1. \ldots \forall x_n. a_1 \wedge \cdots \wedge a_m \Rightarrow a$, where the a_i's are atoms. This results from the syntactic restrictions on pre- and postconditions, together with the weakest preconditions rules which are only using postconditions in negative positions. Therefore we simply need to decide whether a given atom is the consequence of other atoms.

We denote by $H^\star$ the congruence closure of a set H of hypotheses $\{a_1, \ldots, a_m\}$. Obviously $\mathcal{S}(H^\star) = \mathcal{S}(H)$ since no new term is created. $H^\star$ is finite and can be computed as a fixpoint.

Algorithm 1 *For any atom a such that $\mathcal{S}(\{a\}) \subseteq \mathcal{S}(H)$, the following algorithm,* $\texttt{decide}(H, a)$*, decides whether $H \models a$.*

1. *First we compute the congruence closure $H^\star$.*
2. *If a is of the form $t_1 = t_2$, we return* `true` *if $t_1 = t_2 \in H^\star$ and* `false` *otherwise.*
3. *If a is of the form* $\texttt{path}(t_1, t_2)$*, we build a directed graph G whose nodes are the subterms of $H^\star$, as follows:*
   (a) for each pair of nodes t and $\texttt{prev}(t)$ *we add an edge from* $\texttt{prev}(t)$ *to t;*
   (b) for each $\texttt{path}(t_1, t_2) \in H^\star$ *we add an edge from t_1 to t_2;*
   (c) for each $t_1 = t_2 \in H^\star$ we add two edges between t_1 and t_2.
4. *Finally we check whether there is a path from t_1 to t_2 in G.*

Obviously this algorithm terminates since $H^\star$ is finite and thus so is G. We now show soundness and completeness for this algorithm.

Theorem 2. $\texttt{decide}(H, a)$ *returns true if and only if $H \models a$.*

Note: the restriction $\mathcal{S}(\{a\}) \subseteq \mathcal{S}(H)$ can be easily met by adding to H the equalities $t = t$ for any subterm t of a; it was only introduced to simplify the proof above.

4.6 Implementation

We have implemented the whole framework of semi-persistence. The implementation relies on an existing proof obligations generator, Why [10]. This tool takes annotated first-order imperative programs as input and uses a traditional weakest precondition calculus to generate proof obligations. The language we use in this paper is actually a subset of Why's input language. We simply use the imperative aspect to make *cur* a mutable variable. Then the resulting proof obligations are *exactly* the same as those obtained by the constraint synthesis defined in Section 4.2.

The Why tool outputs proof obligations in the native syntax of various existing provers. In particular, these formulas can be sent to Ergo [5], an automatic

prover for first-order logic which combines congruence closure with various built-in decision procedures. We first simply axiomatized theory $\mathcal{T}$ using (A_1)–(A_3), which proved to be powerful enough to verify all examples from this paper and several other benchmark programs. Yet it is possibly incomplete (automatic theorem provers use heuristics to handle quantifiers in first-order logic). To achieve completeness, and to assess the results of Section 4.5, we also implemented theory $\mathcal{T}$ as a new built-in decision procedure in Ergo. Again we verified all the benchmark programs.

5 Conclusion

We have introduced the notion of *semi-persistent* data structures, where update operations are restricted to ancestors of the most recent version. Semi-persistent data structures may be more efficient than their fully persistent counterparts, and are of particular interest in implementing backtracking algorithms. We have proposed an almost automatic way of checking the legal use of semi-persistent data structures. It is based on light user annotations in programs, from which proof obligations are extracted and automatically discharged by a decision procedure.

There is a lot of remaining work to be done. First, the language introduced in Section 3, in which we check for legal use of semi-persistence, could be greatly enriched. Beside the missing features such as polymorphism or recursive datatypes, it would be of particular interest to consider simultaneous use of several semi-persistent data structures and dynamic creation of semi-persistent data structures. Regarding the former, one would probably need to express disjointness of version subtrees, and thus to enrich the logical fragment used in annotations with disjunctions and negations; we may lose decidability of the logic, though. Regarding the latter, it would imply to express in the logic the freshness of the allocated pointers and to maintain the newest versions for each data structures.

Another interesting direction would be to provide systematic techniques to make data structures semi-persistent as previously done for persistence [9]. Clearly what we did for lists could be extended to tree-based data structures. It would be even more interesting to formally verify semi-persistent data structure *implementations*, that is to show that the contents of any ancestor of the version being updated is preserved. Since such implementations are necessarily using imperative features (otherwise they would be fully persistent), proving their correctness requires verification techniques for imperative programs. This could be done for instance using verification tools such as SPEC# [2] or Caduceus [11]. However, we would prefer verifying Ocaml code, as given in the long version of this paper [7] for instance, but unfortunately there is currently no tool to handle such code.

References

1. Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8):145–147, 1991.

2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, number 3362 in LNCS. Springer, 2004.
3. Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, 1999.
4. Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Symposium on Principles of Programming Languages*, pages 25–37, 1998.
5. Sylvain Conchon and Evelyne Contejean. Ergo: A Decision Procedure for Program Verification. `http://ergo.lri.fr/`.
6. Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*, Freiburg, Germany, October 2007.
7. Sylvain Conchon and Jean-Christophe Filliâtre. Semi-Persistent Data Structures. Research Report 1474, LRI, Université Paris Sud, September 2007. `http://www.lri.fr/~filliatr/ftp/publis/spds-rr.pdf`.
8. Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
9. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
10. J.-C. Filliâtre. The Why verification tool. `http://why.lri.fr/`.
11. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification (Tool presentation). In *Proceedings of CAV'2007*, 2007. To appear.
12. John Hannan. A type-based analysis for stack allocation in functional languages. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 172–188, London, UK, 1995. Springer-Verlag.
13. D. E. Knuth. Dancing links. In Bill Roscoe Jim Davies and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 187–214. Palgrave, 2000.
14. J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, pages 28–52, 1998.
15. Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–47, New York, NY, USA, 1983. ACM Press.
16. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
17. Silvio Ranise and Calogero Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 206–215, Washington, DC, USA, 2006. IEEE Computer Society.
18. Frances Spalding and David Walker. Certifying compilation for a language with stack allocation. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 407–416, Washington, DC, USA, 2005. IEEE Computer Society.
19. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, 1994.