Semi-Persistent Data Structures

Sylvain Conchon Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Parc Orsay Université, F-91893
{conchon,filliatr}@lri.fr

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. Additionally, we give some examples of semi-persistent data structures (arrays, lists and hash tables).

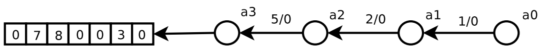
Keywords Persistent Data Structure, Backtracking

1. Introduction

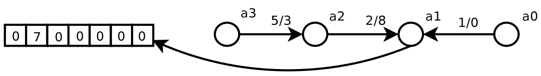
A data structure is said to be *persistent* when any update operation returns a new structure without altering the old version. In purely applicative programming, data structures are automatically persistent (Okasaki 1998). Yet this notion is more general and the exact meaning of persistent is *observationally immutable*. Driscoll et al. even proposed systematic techniques to make imperative data structures persistent (Driscoll et al. 1989). 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.

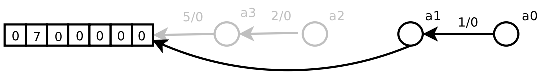
A semi-persistent data structure can be more efficient than its fully persistent counterpart, both in time and space. Let us illustrate this fact on an example. We consider persistent arrays as introduced by Baker (Baker 1991). The basic idea is to use an imperative array for the newest version of the persistent array and indirections for old versions. For instance, starting with an array a_0 initialized with 0, and performing the successive updates $a_1 = \mathtt{set}(a_0, 1, 7)$, $a_2 = \mathtt{set}(a_1, 2, 8)$ and $a_3 = \mathtt{set}(a_2, 5, 3)$, 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, and more efficient, data structure. An Objective Caml code for such semi-persistent arrays is given in Section 4.1. On this particular example, we see that the data structure behaves like an "undo stack" (the pairs index/value stored in indirections are precisely the undo operations). The same idea could be applied to any imperative data structure where update operations can be undone. Contrary to an undo stack which messes the backtracking algorithm, this one is actually hidden in the data structure. Thus the algorithm 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*.

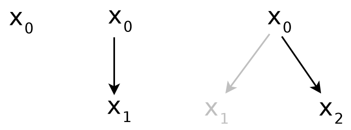
This article only addresses the former point. Regarding the latter, we simply give implementations of semi-persistent data structures. Proving the correctness of these implementations is out of the scope of this paper.

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 proof obligations which express the correctness of programs using a weakest precondition-like calculus (Dijkstra 1976). These obligations appear to be 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.

To make it more precise, let us consider two programs manipulating a semi-persistent data structure. Each program is a function taking a version x_0 of the data structure as argument and generating new versions using an update operation upd:

fun $f_1\ x_0 =$	fun $f_2\ x_0 =$
$\{$valid$(x_0)\}$	$\{$valid$(x_0)\}$
let $x_1 =$ upd x_0 in	let $x_1 =$ upd x_0 in
let $x_2 =$ upd x_0 in	let $x_2 =$ upd x_0 in
upd x_2	upd x_1

Each function has a precondition requiring the validity of x_0 (through a valid predicate). Then both functions successively build two new versions x_1 and x_2 as successors of x_0. This can be illustrated by the following version trees, where arrows represent version successors and a gray arrow stands for a version that has become invalid:



Finally, function f_1 updates x_2, which is legal; and function f_2 updates x_1, which is illegal since x_1 is not an ancestor of the newest version x_2. If the x_i were semi-persistent arrays, as described above, the contents of x_1 would indeed be *modified* by the creation of x_2. Thus x_1 should not be accessed anymore.

As illustrated by this example, evaluation order matters as the creation of a new version has side-effects on other versions. Thus semi-persistence is clearly related to impure data structures. For this reason, our weakest precondition calculus relies on a small effect inference which detects the mutation of the data structure.

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 (Hannan 1995; Blanchet 1998) 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 f_2 above. Second, works on stack analysis to ensure memory safety (Morrisett et al. 1998; Spalding and Walker 2005; Tofte and Talpin 1994) 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 (Nelson 1983; Benedikt et al. 1999; Ranise and Zarba 2006). However, our logic is much simpler and thus we provide a dedicated decision procedure. Finally, we can mention Knuth's *dancing links* (Knuth 2000) as an example of a data structure whose efficiency is improved when used in a backtracking algorithm; 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. Section 2 introduces a small programming language to manipulate semi-persistent data structures, as a support for our formalization. Then Section 3 defines the proof system which checks the valid use of semi-persistent data structures. Finally Section 4 gives examples of semi-persistent data structures and shows the benefits of semi-persistence with some benchmarks.

2. 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, through a single update operation upd. Multiple data structures and dynamic allocation is discussed in Section 5.

2.1 Syntax

The syntax of our language is as follows:

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

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 the operation upd. 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 bool of booleans or any other uninterpreted type (δ). The syntax of pre- and postconditions will be given later in Section 3. A program Δ is a finite set of mutually recursive functions.

2.2 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 $p_1, \dots, p_m$ of pointers, p_m being the top of the stack. The semantics is straightforward, except for operation upd. The latter 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 a new value p is pushed, which is also the result of the operation. Note that reduction of upd p_n is blocked whenever p_n is not an element of S, which is precisely what we intend to prevent.

There is an obvious property of reduction which will be useful in Section 3.4:

LEMMA 1. *The first element of a state is preserved by reduction, that is if $e, pS \to e', S'$ then $S' = pS''$.*

2.3 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:

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

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

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

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

Figure 1. Operational Semantics

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 a type declaration for upd, as follows:

$$\mathtt{upd} : (x : \mathtt{semi}) \rightarrow^{\top} \{\phi_{\mathtt{upd}}\}\ \mathtt{semi}\ \{\psi_{\mathtt{upd}}\}$$

where $\phi_{\mathtt{upd}}$ and $\psi_{\mathtt{upd}}$ are given later. Given a typing environment Γ, the judgment $\Gamma \vdash e : \iota, \epsilon$ means "e is 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 3.1. Note that there is no typing rule for pointers, to prevent their explicit use in programs.

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.

2.4 Examples

Let us consider the two functions f_1 and f_2 from the introduction ($\mathtt{valid}(x_0)$ being now expressed as $\mathtt{path}(x_0, cur)$), that is

$$
\begin{array}{l|l}
\mathtt{fun}\ f_1\ (x_0 : \mathtt{semi}) = & \mathtt{fun}\ f_2\ (x_0 : \mathtt{semi}) = \\
\quad \{\mathtt{path}(x_0, cur)\} & \quad \{\mathtt{path}(x_0, cur)\} \\
\quad \mathtt{let}\ x_1 = \mathtt{upd}\ x_0\ \mathtt{in} & \quad \mathtt{let}\ x_1 = \mathtt{upd}\ x_0\ \mathtt{in} \\
\quad \mathtt{let}\ x_2 = \mathtt{upd}\ x_0\ \mathtt{in} & \quad \mathtt{let}\ x_2 = \mathtt{upd}\ x_0\ \mathtt{in} \\
\quad \mathtt{upd}\ x_2 & \quad \mathtt{upd}\ x_1
\end{array}
$$

Let S be a state composed of a single pointer p. The reduction of $f_1\ p$ in S runs as follows:

$$
\begin{array}{ll}
f_1\ p, p & \rightarrow \mathtt{let}\ x_1 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ x_2, p \\
& \rightarrow \mathtt{let}\ x_1 = p_1\ \mathtt{in}\ \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ x_2, pp_1 \\
& \rightarrow \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ x_2, pp_1 \\
& \rightarrow \mathtt{let}\ x_2 = p_2\ \mathtt{in}\ \mathtt{upd}\ x_2, pp_2 \\
& \rightarrow \mathtt{upd}\ p_2, pp_2 \\
& \rightarrow p_3, pp_2p_3
\end{array}
$$

and ends on the value p_3. On the contrary, the reduction of $f_2\ p$ in S runs as follows:

$$
\begin{array}{ll}
f_2\ p, p & \rightarrow \mathtt{let}\ x_1 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ x_1, p \\
& \rightarrow \mathtt{let}\ x_1 = p_1\ \mathtt{in}\ \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ x_1, pp_1 \\
& \rightarrow \mathtt{let}\ x_2 = \mathtt{upd}\ p\ \mathtt{in}\ \mathtt{upd}\ p_1, pp_1 \\
& \rightarrow \mathtt{let}\ x_2 = p_2\ \mathtt{in}\ \mathtt{upd}\ p_1, pp_2 \\
& \rightarrow \mathtt{upd}\ p_1, pp_2
\end{array}
$$

and blocks on $\mathtt{upd}\ p_1$.

3. 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.

3.1 Theory of Semi-Persistence

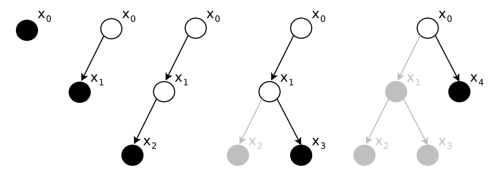
The syntax of annotations is as follows:

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

Terms are built from variables, pointers and a single function symbol pred. 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 3.5 when introducing the decision procedure. In the remaining 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 pred has signature $\mathtt{semi} \rightarrow \mathtt{semi}$. Formulae can only refer to variables of type semi. Additionally, a function postcondition may refer to the function result, represented by the variable r. 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. The following figure illustrates the successive version trees for the sequence of declarations $x_1 = \mathtt{upd}\ x_0$, $x_2 = \mathtt{upd}\ x_1$, $x_3 = \mathtt{upd}\ x_1$ and $x_4 = \mathtt{upd}\ x_0$:



The newest version is pictured as a black node, valid versions as white nodes and invalid ones as gray nodes. We use pred and path to define the notion of ancestor: $\mathtt{pred}(x)$ is the immediate ancestor of x and $\mathtt{path}(x, y)$ holds whenever x is an ancestor of y. The corresponding theory can be axiomatized as follows:

$$\textsc{Var}\ \frac{x:\iota\in\Gamma}{\Gamma\vdash x:\iota,\bot}\qquad \textsc{Const}\ \frac{c:\iota\in\Gamma}{\Gamma\vdash c:\iota,\bot}\qquad \textsc{App}\ \frac{f:(x:\iota_1)\rightarrow^{\epsilon_2}\{\phi\}\,\iota_2\,\{\psi\}\in\Gamma\qquad \Gamma\vdash e:\iota_1,\epsilon_1}{\Gamma\vdash f\ e:\iota_2,\epsilon_1\vee\epsilon_2}$$

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

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

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

Figure 2. Typing Rules

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) $\forall xy.\,\mathtt{path}(x,\mathtt{pred}(y))\Rightarrow\mathtt{path}(x,y)$
(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 $\mathcal{T}$.

Note that the three axioms (A_1)–(A_3) exactly define path as the reflexive transitive closure of $\mathtt{pred}^{-1}$, 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).

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 upd function, whose type reads as follows:

$$(x:\mathtt{semi})\rightarrow^{\top}\{\mathtt{path}(x,\mathit{cur})\}\ \mathtt{semi}\ \{\mathtt{pred}(\mathtt{r})=x\wedge\mathit{cur}=\mathtt{r}\}$$

upd's precondition requires x to be a valid version, that is an ancestor of the newest version *cur*, which is expressed by $\mathtt{path}(x,\mathit{cur})$. upd has effect $\top$, to indicate the modification of *cur*. upd's postcondition is two-fold: first, it says that the returned value r is an immediate successor of x; second, it states that the new value of *cur* is precisely r.

We are now able to define the judgements used in Section 2.3 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 $\Gamma,\mathit{cur}:\mathtt{semi},\mathtt{r}:\iota\vdash\psi$ when return type ι is semi and for $\Gamma,\mathit{cur}:\mathtt{semi}\vdash\psi$ otherwise. It is important to notice that function preconditions may only refer to the function argument and *cur*, and that postconditions may additionally refer to r.

In Section 3.5, we will need the following subterms property:

LEMMA 2. *If H is a set of atoms, t_1 a subterm of H and $H\models\mathtt{path}(t_1,t_2)$ then t_2 is a subterm of H.*

3.2 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 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 $\mathcal{C}$ is given in Figure 3. The only non trivial case is function call, for which we introduce a formula transformer $\mathtt{frame}_f(\phi)$ to deal with the frame problem. There are two cases depending on whether f has an effect or not; then we get for free the invariance of *cur* when f does not modify it (note that it could have been expressed using an equality).

The formula established by an expression may be weakened, as stated by the following lemma:

LEMMA 3. *If $\models\forall\mathtt{r}.\forall\mathit{cur}.\,\phi_1\Rightarrow\phi_2$ and $\models\mathcal{C}(e,\phi_1)$ then $\models\mathcal{C}(e,\phi_2)$.*

3.3 Examples

This section details the constraints obtained on several program examples.

3.3.1 Simple Examples

Let us consider again the two functions f_1 and f_2 from Section 2.4 and compute the associated constraints for an empty postcondition (written true in the following and which could stand for the tautology $\mathit{cur}=\mathit{cur}$). The constraint $\mathcal{C}(f_1)$ is

$$\begin{array}{l}\forall x_0.\,\forall\mathit{cur}.\,\mathtt{path}(x_0,\mathit{cur})\Rightarrow\\ \quad\mathtt{path}(x_0,\mathit{cur})\wedge\\ \quad\forall x_1.\,\forall\mathit{cur}_1.\,(\mathtt{pred}(x_1)=x_0\wedge\mathit{cur}_1=x_1)\Rightarrow\\ \quad\quad\mathtt{path}(x_0,\mathit{cur}_1)\wedge\\ \quad\quad\forall x_2.\,\forall\mathit{cur}_2.\,(\mathtt{pred}(x_2)=x_0\wedge\mathit{cur}_2=x_2)\Rightarrow\\ \quad\quad\quad\mathtt{path}(x_2,\mathit{cur}_2)\wedge\\ \quad\quad\quad\forall x_3.\,\forall\mathit{cur}_3.\,(\mathtt{pred}(x_3)=x_2\wedge\mathit{cur}_3=x_3)\Rightarrow\mathtt{true}\end{array}$$

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

$$\begin{array}{rl}\mathtt{path}(x_0,\mathit{cur}) & \vdash\mathtt{path}(x_0,\mathit{cur})\\ \mathtt{path}(x_0,\mathit{cur}),\mathtt{pred}(x_1)=x_0,\mathit{cur}_1=x_1 & \vdash\mathtt{path}(x_0,\mathit{cur}_1)\\ \mathtt{path}(x_0,\mathit{cur}),\mathtt{pred}(x_1)=x_0,\\ \mathit{cur}_1=x_1,\mathtt{pred}(x_2)=x_0,\mathit{cur}_2=x_2 & \vdash\mathtt{path}(x_2,\mathit{cur}_2)\end{array}$$

The three of them hold in theory $\mathcal{T}$ and thus f_1 is correct. Similarly, the constraint $\mathcal{C}(f_2)$ can be computed and split into three proof obligations. The first two are exactly the same as for f_1 but the third one is slightly different:

$$\begin{array}{rl}\mathtt{path}(x_0,\mathit{cur}),\mathtt{pred}(x_1)=x_0,\\ \mathit{cur}_1=x_1,\mathtt{pred}(x_2)=x_0,\mathit{cur}_2=x_2 & \vdash\mathtt{path}(x_1,\mathit{cur}_2)\end{array}$$

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

3.3.2 Backtracking Example

As a more complex example, let us consider a backtracking algorithm. The pattern of a program performing backtracking on a per-

$$\begin{array}{rcl}
\mathtt{frame}_f(\phi) & = & \phi_f\{x \leftarrow \mathtt{r}\} \wedge \forall \mathtt{r}'.\, \psi_f\{\mathtt{r} \leftarrow \mathtt{r}', x \leftarrow \mathtt{r}\} \Rightarrow \phi\{\mathtt{r} \leftarrow \mathtt{r}'\} \\
 & & \text{if } f : (x : \iota) \rightarrow^{\perp} \{\phi_f\}\, \iota'\, \{\psi_f\} \\
\mathtt{frame}_f(\phi) & = & \phi_f\{x \leftarrow \mathtt{r}\} \wedge \forall \mathtt{r}'\, cur'.\, \psi_f\{\mathtt{r} \leftarrow \mathtt{r}', x \leftarrow \mathtt{r}, cur \leftarrow cur'\} \Rightarrow \phi\{\mathtt{r} \leftarrow \mathtt{r}', cur \leftarrow cur'\} \\
 & & \text{if } f : (x : \iota) \rightarrow^{\top} \{\phi_f\}\, \iota'\, \{\psi_f\}
\end{array}$$

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

Figure 3. Constraints synthesis

sistent data structure is a recursive function bt looking like

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

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 $bt(\mathtt{upd}\ x)$, which can be written as follows:

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

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 $bt(\mathtt{upd}\ x)$ 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:

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

The constraint $\mathcal{C}(bt)$ is

$$\begin{array}{l}
\forall x. \forall cur.\, \mathtt{path}(x, cur) \Rightarrow \\
\quad \mathtt{path}(x, cur) \wedge \\
\quad \forall x_1. \forall cur_1.\, (\mathtt{pred}(x_1) = x \wedge cur_1 = x_1) \Rightarrow \\
\quad\quad \mathtt{path}(x_1, cur_1) \wedge \\
\quad\quad \forall x_2. \forall cur_2.\, \mathtt{path}(x_1, cur_2) \Rightarrow \\
\quad\quad\quad \mathtt{path}(x, cur_2) \wedge \\
\quad\quad\quad \forall x_3. \forall cur_3.\, (\mathtt{pred}(x_3) = x \wedge cur_3 = x_3) \Rightarrow \\
\quad\quad\quad\quad \mathtt{path}(x_3, cur_3) \wedge \\
\quad\quad\quad\quad \forall x_4. \forall cur_4.\, \mathtt{path}(x_3, cur_4) \Rightarrow \mathtt{path}(x, cur_4)
\end{array}$$

which is valid in theory $\mathcal{T}$.

3.4 Soundness

In the remaining 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*

$$\left\{ \begin{array}{l} \mathit{pred}(p_{i+1}) = p_i \text{ for all } 1 \leq i < n \\ cur = p_n \end{array} \right.$$

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

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

LEMMA 4. $S \models \mathcal{C}(E[e], \phi)$ *if and only if* $S \models \mathcal{C}(e, E[\phi])$.

PROOF. Immediate by induction on E. □

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 $\phi \equiv \mathtt{pred}(p_2) = p_1$, then $\mathcal{C}(e, \phi)$ is

$$\begin{array}{l}
\mathtt{path}(p_1, cur) \wedge \\
\forall \mathtt{r}'\, cur'.\, (\mathtt{pred}(r') = p_1 \wedge cur' = \mathtt{r}') \Rightarrow \mathtt{pred}(p_2) = p_1
\end{array}$$

which holds in S. But $S' \equiv p_1 p$ for a fresh p, $e' \equiv p$, and $\mathcal{C}(e', \phi)$ is $\mathtt{pred}(p_2) = p_1$ 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. Now, for a function declaration, both its body and its specification may only refer to its argument x. Therefore we are only considering formulae $\mathcal{C}(e, \phi)$ referring to at most one pointer, which is the value of x. This excludes the formula $\mathtt{pred}(p_2) = p_1$ in the example above.

In the remaining of this section, we only consider program expressions and formulae which only refer to a single pointer p (representing the function argument). For technical reasons, it is also convenient to restrict ourselves to states whose p is the first element. This is not a restriction, as stated by the following lemma:

LEMMA 5. *Let e be an expression and ϕ a formula which both refer to a single pointer p. Then, for any state $S_1 p S_2$, $S_1 p S_2 \models \mathcal{C}(e, \phi)$ if and only if $p S_2 \models \mathcal{C}(e, \phi)$.*

We are now able to prove preservation of validity:

LEMMA 6. *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 $S' \models \mathcal{C}(e', \phi)$.*

PROOF. The proof is done by induction on e:

- If $e = \mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2$ then $e' = e_1$ and $S' = S$. We have $\mathcal{C}(e, \phi) = \mathcal{C}(e_1, \phi) \wedge \mathcal{C}(e_2, \phi)$ and thus $S' \models \mathcal{C}(e', \phi)$. Similarly for `if false`.
- If $e = \mathtt{let}\ x = v\ \mathtt{in}\, e_1 e$ then $e' = e_1\{x \leftarrow v\}$ and $S' = S$. We have $\mathcal{C}(e, \phi) = \mathcal{C}(v, \mathcal{C}(e_1, \phi)\{x \leftarrow \mathtt{r}\}) = \mathcal{C}(e_1, \phi)\{x \leftarrow v\} = \mathcal{C}(e_1\{x \leftarrow v\}, \phi)$, hence the result holds.

- If $e = f\ v$ then $e' = e_f\{x \leftarrow v\}$ and $S' = S$. We have $\mathcal{C}(e, \phi) = \mathcal{C}(v, \mathtt{frame}_f(\phi))$ which is of the form $\phi_f\{x \leftarrow v\} \wedge \forall r'. \psi_f\{x \leftarrow v, \dots\} \Rightarrow \phi\{x \leftarrow v, \dots\}$. Since f is correct, we have $\models \mathcal{C}(f) = \forall x. \phi_f \Rightarrow \mathcal{C}(e_f, \psi_f)$. Thus $S \models \phi_f\{x \leftarrow v\} \Rightarrow \mathcal{C}(e_f\{x \leftarrow v\}, \psi_f\{x \leftarrow v\})$ and the result follows by Lemma 3.
- If $e = \mathtt{upd}\ p_n$ then $S = p_1 \cdots p_n p_{n+1} \cdots p_m$, $e' = p$ and $S' = p_1 \cdots p_n p$. We have $S \models \mathcal{C}(\mathtt{upd}\ p_n, \phi)$, that is $S \models \cdots \wedge \forall r' cur'. (\mathtt{pred}(r') = p_n \wedge cur' = r') \Rightarrow \phi\{\mathtt{r} \leftarrow r', cur \leftarrow cur'\}$. By instantiating both r' and cur' by p, we have $S \models \mathtt{pred}(p) = p_n \Rightarrow \phi\{\mathtt{r} \leftarrow p, cur \leftarrow p\}$. Since this formula only refers to the first element of S, which is also by Lemma 1 the first element of S', we have $S' \models \mathtt{pred}(p) = p_n \Rightarrow \phi\{\mathtt{r} \leftarrow p, cur \leftarrow p\}$ and thus $S' \models \phi\{\mathtt{r} \leftarrow p, cur \leftarrow p\}$, that is $S' \models \mathcal{C}(p, \phi)$.
- If $e = E[e_1]$ and $E[e_1], S \rightarrow E[e_1'], S'$ with $e_1, S \rightarrow s_1', S'$, then $S \models \mathcal{C}(e_1, E[\phi])$ by Lemma 4. By induction hypothesis on e_1 we have $S' \models \mathcal{C}(e_1', E[\phi])$. Then again by Lemma 4 we have $S' \models \mathcal{C}(E[e'1], \phi)$.

□

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.*

PROOF. The proof is done by induction on the length of the evaluation $\rightarrow^*$ and by case analysis on the last reduction:

- If e is irreducible then it is either a value v and the result holds, or it is an expression $E[\mathtt{upd}\, p_n]$ with $p_n \notin S$. By hypothesis, we have $S \models \mathcal{C}(E[\mathtt{upd}\, p_n], \phi)$ and thus $S \models \mathcal{C}(\mathtt{upd}\, p_n, E[\phi])$ by Lemma 4. Since $\mathcal{C}(\mathtt{upd}\, p_n, E[\phi]) = \mathtt{path}(p_n, cur) \wedge \dots$, it contradicts $p_n \notin S$.
- If $e, S \rightarrow e_1, S_1 \rightarrow^* e', S' \not\rightarrow$ then by Lemma 6 $S_1 \models \mathcal{C}(e_1, \phi)$ and the result holds by induction hypothesis on e_1.

□

3.5 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

$$\forall x_1. \dots \forall x_n. a_1 \wedge \dots \wedge a_m \Rightarrow a$$

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.

Let H be the set of hypotheses $\{a_1, \dots, a_m\}$. For any atom a such that $\mathcal{S}(\{a\}) \subseteq \mathcal{S}(H)$, the following algorithm, $\mathtt{decide}(H, a)$, decides whether $H \models a$.

1. first we replace H by the congruence closure $H^\star$ of H, that is the smallest set of atoms such that
   - if $a \in H$ then $a \in H^\star$;
   - if $t \in \mathcal{S}(H^\star)$ then $t = t \in H^\star$;
   - if $t_1 = t_2 \in H^\star$ then $t_2 = t_1 \in H^\star$;
   - if $t_1 = t_2 \in H^\star$ and $t_2 = t_3 \in H^\star$ then $t_1 = t_3 \in H^\star$;
   - if $t_1 = t_2 \in H^\star$ and $\mathtt{pred}(t_1) \in \mathcal{S}(H^\star)$ then $\mathtt{pred}(t_1) = \mathtt{pred}(t_2) \in H^\star$;
   - if $t_1 = t_2 \in H^\star$, $t_3 = t_4 \in H^\star$ and $\mathtt{path}(t_1, t_3) \in H^\star$ then $\mathtt{path}(t_2, t_4) \in H^\star$.
2. if a is of the form $t_1 = t_2$, we return $\mathtt{true}$ if $t_1 = t_2 \in H^\star$ and $\mathtt{false}$ otherwise;
3. if a is of the form $\mathtt{path}(t_1, t_2)$, we build a directed graph G whose nodes are the subterms of $H^\star$ and a;
4. for each pair of nodes t and $\mathtt{pred}(t)$ we add an edge from $\mathtt{pred}(t)$ to t;
5. for each $\mathtt{path}(t_1, t_2) \in H^\star$ we add an edge from t_1 to t_2;
6. for each $t_1 = t_2 \in H^\star$ we add two edges between t_1 and t_2;
7. finally we check whether there is a path from t_1 to t_2 in G.

This algorithm is sound and complete:

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

PROOF. *(Soundness)* Let $\mathcal{M}$ be a model of H and let us show that $\mathcal{M}$ is a model of a. Obviously, $\mathcal{M}$ is also a model of $H^\star$. If a is $\mathtt{path}(t_1, t_2)$ then the proof is by induction on the length of the path. If the path has length 0, then $t_1 \equiv t_2$ and the result holds since $\mathcal{M}$ is a model of axiom (A_1). Otherwise, we proceed by case analysis on the last step of the path since $\mathcal{M}$ is a model of the transitivity axiom (A_3). If the last step is an edge from $\mathtt{pred}(t)$ to t then the result holds since $\mathcal{M}$ is a model of both (A_1) and (A_2). If the last step is an edge corresponding to $\mathtt{path}(t_3, t_4) \in H^\star$ then the results holds since $\mathcal{M}$ is a model of $H^\star$. If the last step is an edge corresponding to $t_3 = t_4 \in H^\star$, then $\mathcal{M}$ is a model of $\mathtt{path}(t_3, t_4)$ by axiom (A_1) and axioms of equality.

(Completeness) Let us assume that $H \models a$. If a is $t_1 = t_2$ then $t_1 = t_2 \in H^\star$ by definition of the congruence closure, and thus the algorithm returns $\mathtt{true}$. If a is $\mathtt{path}(t_1, t_2)$ we proceed by induction on the proof of $H \models a$ and by case analysis on the last rule:

- (A_1): There is an empty path from t_1 to t_2;
- (A_2): By induction hypothesis there is a path from t_1 to $\mathtt{pred}(t_2)$ and thus a path from t_1 to t_2;
- (A_3): We have $H \models \mathtt{path}(t_1, t)$ and $H \models \mathtt{path}(t, t_2)$. By Lemma 2, we have $t \in \mathcal{S}(H)$ and thus we can apply the induction hypothesis.
- congruence: We have $H \models u_1 = t_1$, $H \models u_2 = t_2$ and $H \models \mathtt{path}(u_1, u_2)$. Thus we have $H \models \mathtt{path}(u_1, t_1)$ and $H \models \mathtt{path}(u_2, t_2)$ (as above) and $H \models \mathtt{path}(u_1, u_2)$ by induction hypothesis; the result follows.

□

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.

3.6 Implementation

We have implemented the whole framework of semi-persistence. The implementation relies on an existing proof obligations generator, Why (Filliâtre). This tool takes annotated first-order imperative programs as input and uses a traditional weakest precondition calculus to generate proof obligations in the native syntax of various existing provers. In particular, proof obligations can be sent to Ergo (Conchon and Contejean), 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 3.5, we also implemented theory $\mathcal{T}$ as a new built-in decision procedure in Ergo. Again we verified all the benchmark programs.

4. Examples of Semi-Persistent Data Structures

We present the implementation of semi-persistent arrays, lists and hash tables and benchmarks to show the benefits of semi-persistence. All these examples are written in Objective Caml (leroy et al. 2005).

4.1 Arrays

First we implement semi-persistent arrays, as described in the introduction of this paper. The interface is similar to persistent arrays:

```
type α t
val create : int → α → α t
val get : α t → int → α
val set : α t → int → α → α t
```

Type α `t` is the polymorphic type of semi-persistent arrays containing values of type α. `create` n v creates a new array of size n with all cells initialized with v. `get` and `set` are the usual access and update functions.

A semi-persistent array is a reference containing a value of type α `data`, which is either an imperative array from Ocaml's standard library or an indirection:

```
type α t = α data ref
and α data =
  | Newest of α array
  | Diff of int × α × α t
```

`Newest` a stands for the newest version of the structure, and `Diff` (i, v, a) stands for a semi-persistent array sharing the contents of a, except at index i where its value is v.

Creation of a new semi-persistent array is simply an allocation of a new reference containing a new imperative array:

```
let create n v = ref (Newest (Array.create n v))
```

To implement `get` and `set`, we first need to implement the rerooting operation, which backtracks to a given version. In this semi-persistent implementation, it simply amounts to writing the values in `Diff` nodes into the imperative array:

```
let rec reroot t = match !t with
  | Newest _ →
      ()
  | Diff (i, v, t') →
      reroot t';
      let Newest a as n = !t' in
      a.(i) ← v;
      t := n
```

A call `reroot` t ensures that t is now the newest version (and thus a direct access to the imperative array); hence we can safely match `!t'` against the `Newest` constructor after the recursive call. Note that `reroot` could be written in CPS to avoid a possible stack overflow.

Then both `get` and `set` start with a call to `reroot` and thus safely ignore the case of an indirection:

```
let get t i =
  reroot t;
  let Newest a = !t in a.(i)

let set t i v =
  reroot t;
  let Newest a as n = !t in
  let old = a.(i) in
  a.(i) ← v;
  let res = ref n in
  t := Diff (i, old, res);
  res
```

The only difference with the fully persistent implementation is in function `reroot`, where the sequence `a.(i) ← v; t := n` would be replaced by the more complex code:

```
 let v' = a.(i) in
 a.(i) ← v;
 t := n;
 t' := Diff (i, v', t)
```

Thus we save, in the semi-persistent implementation, an array access, a constructor allocation and an assignment.

4.2 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.

To implement semi-persistent lists, we use the following type of double-linked lists:

```
type α t = { mutable head : α;
             tail : α t;
             mutable pc : α t }
```

Fields `head` and `tail` are the usual fields for linked lists, except that `head` is a mutable field to allow further modification of its contents. The additional field `pc` keeps track of a previous cons, if any.

Similarly to the constant `null` in C or Java programs, we would like to represent the empty list as a statically allocated constant value of type α `t`. Since there is no way to build such a polymorphic value (we can not instantiate the `head` field), we turn the empty list into a function taking a dummy value as argument:

```
let nil v =
  let rec null =
    { head = v; tail = null; pc = null }
  in null
```

This function allocates a single cons cell `null` where `tail` and `pc` fields are physically equal to `null`. Thus testing for the empty list is as simple as

```
let is_nil l = l.tail == l
```

Additionally, the empty list is used as the value for `pc` when there is no previous cons.

The `cons` function checks the existence of a previously allocated block (testing `pc` for being the empty list) and reuses it if any:

```
let cons x l =
  if is_nil l.pc then begin
    let n = { l with v=x; tail=l } in
    if not (is_nil l) then l.pc ← n;
    n
  end else begin
```

```
    let c = l.pc in
    c.v ← x;
    c
  end
```

The test `not (is_nil l)` prevents from reusing the successor of the empty list, since it is shared by all lists.

It is worth noticing that such semi-persistent lists could even be used to improve the implementation of semi-persistent arrays from the previous section, since `Diff` cells could similarly be reused.

4.3 Hash Tables

Combining (semi-)persistent arrays with (semi-)persistent lists, one easily gets (semi-)persistent hash tables. Such a combination is nicely implemented as a module parameterized with data structures for arrays and lists respectively. Introducing the following signatures for these parameters:

```
module type Array = sig
  type α t
  val create : int → α → α t
  val get : α t → int → α
  val set : α t → int → α → α t
end
module type List = sig
  type α t
  val nil : α → α t
  val cons : α → α t → α t
end
```

the generic implementation of hash tables is as follows:

```
module MakeHT(PA : Array)(IL : List) = struct
  type α t = { size : int; data : α IL.t PA.t }

  let create n v =
    { size = n; data = PA.create n (IL.nil v) }

  let add h x =
    let i = x mod h.size in
    { h with data =
        PA.set h.data i
          (IL.cons x (PA.get h.data i)) }
end
```

By instantiating arrays and lists implementations by semi-persistent arrays and lists from Sections 4.1 and 4.2 respectively, we automatically get semi-persistent hash tables.

4.4 Benchmarks

We present some benchmarks to show the benefits of semi-persistence. Each of the previous three data structures 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 successive update operations are performed on the data structure between two branchings points (that is `set` on arrays, `cons` on lists and `add` on hash tables). It results in a total of $4^6 N$ elementary operations and $3(4^6 - 1)$ backtracks. 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.

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 to implement 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 on which to 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 the following extensions:

- *handling multiple operations*: We have restricted our study to a single update operation, but supporting several access or update operations is straightforward. Indeed, it simply amounts to extending the operational semantics and assigning types to these operations. As update operations, access operations require the version to be valid. However, they may or may not invalidate the successors of their argument. For instance, the type of the `get` operation given in Section 4.1 would look like

  $$\texttt{get} : (x : \alpha\,\texttt{t}) \times \texttt{int} \rightarrow^{\top} \{\texttt{path}(x, cur)\}\, \alpha\, \{cur = x\}$$

  since this function backtracks to version x with a call to `reroot`. If it was accessing x without any modification, we would give it the following type instead:

  $$\texttt{get} : (x : \alpha\,\texttt{t}) \times \texttt{int} \rightarrow^{\perp} \{\texttt{path}(x, cur)\}\, \alpha\, \{\}$$

- *simultaneous use of several semi-persistent data structures*: 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.
- *dynamic creation of semi-persistent data structures*: This 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 (Driscoll et al. 1989). 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# (Barnett et al. 2004) or Caduceus (Filliâtre and Marché 2007). However, we would prefer verifying Ocaml code, as given in Section 4 for instance, but unfortunately there is currently no tool to handle such code.

References

Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8):145–147, 1991. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/122598.122614.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, number 3362 in LNCS. Springer, 2004.

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.

Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Symposium on Principles of Programming Languages*, pages 25–37, 1998. URL `citeseer.ist.psu.edu/blanchet98escape.html`.

Sylvain Conchon and Evelyne Contejean. Ergo: A Decision Procedure for Program Verification. `http://ergo.lri.fr/`.

Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

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.

J.-C. Filliâtre. The Why verification tool. `http://why.lri.fr/`.

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.

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. ISBN 3-540-60360-3.

D.E. Knuth. Dancing links. In Bill Roscoe Jim Davies and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 187–214. Palgrave, 2000.

Xavier leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml System, 2005. `http://caml.inria.fr/`.

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, pages 28–52, 1998. URL `citeseer.ist.psu.edu/article/morrisett98stackbased.html`.

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. ISBN 0-89791-090-7. doi: http://doi.acm.org/10.1145/567067.567073.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

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. ISBN 0-7695-2678-0. doi: http://dx.doi.org/10.1109/SEFM.2006.7.

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. ISBN 0-7695-2266-1. doi: http://dx.doi.org/10.1109/LICS.2005.9.

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. URL `citeseer.ist.psu.edu/tofte94implementation.html`.