How to make ad hoc proof automation less ad hoc

GEORGES GONTHIER
Microsoft Research, Cambridge, United Kingdom
(e-mail: gonthier@microsoft.com)

BETA ZILIANI
Max Planck Institute for Software Systems (MPI-SWS), Saarbrücken, Germany
(e-mail: beta@mpi-sws.org)

ALEKSANDAR NANEVSKI
IMDEA Software Institute, Madrid, Spain
(e-mail: aleks.nanevski@imdea.org)

DEREK DREYER
Max Planck Institute for Software Systems (MPI-SWS), Saarbrücken, Germany
(e-mail: dreyer@mpi-sws.org)

Abstract

Most interactive theorem provers provide support for some form of user-customizable proof automation. In a number of popular systems, such as Coq and Isabelle, this automation is achieved primarily through tactics, which are programmed in a separate language from that of the prover’s base logic. While tactics are clearly useful in practice, they can be difficult to maintain and compose because, unlike lemmas, their behavior cannot be specified within the expressive type system of the prover itself.

We propose a novel approach to proof automation in Coq that allows the user to specify the behavior of custom automated routines in terms of Coq’s own type system. Our approach involves a sophisticated application of Coq’s canonical structures, which generalize Haskell type classes and facilitate a flexible style of dependently-typed logic programming. Specifically, just as Haskell type classes are used to infer the canonical implementation of an overloaded term at a given type, canonical structures can be used to infer the canonical proof of an overloaded lemma for a given instantiation of its parameters. We present a series of design patterns for canonical structure programming that enable one to carefully and predictably coax Coq’s type inference engine into triggering the execution of user-supplied algorithms during unification, and we illustrate these patterns through several realistic examples drawn from Hoare Type Theory. We assume no prior knowledge of Coq and describe the relevant aspects of Coq type inference from first principles.

1 Introduction

In recent years, interactive theorem proving has been successfully applied to the verification of important mathematical theorems and substantial software code bases. Some of the most significant examples are the proof of the Four Color Theorem (Gonthier, 2008) (in Coq), the verification of the optimizing compiler CompCert (Leroy, 2009) (also in
Coq), and the verification of the operating system microkernel seL4 (Klein et al., 2010) (in Isabelle). The interactive theorem provers (a.k.a. proof assistants) employed in these verification efforts depend on higher-order logics and type systems in order to maximize expressiveness and generality, but also to facilitate modularity and reuse of proofs. However, despite the expressiveness of these theorem provers, effective solutions to some verification problems can often only be achieved by going outside of the provers’ base logics.

To illustrate, consider the following Coq lemma, which naturally arises when reasoning about heaps and pointer aliasing:

\[
\text{noalias} : \forall h : \text{heap}. \forall x_1, x_2 : \text{ptr}. \forall v_1 : A_1. \forall v_2 : A_2. \\
\text{def } (x_1 \mapsto v_1 \cdot x_2 \mapsto v_2 \cdot h) \rightarrow x_1 \neq x_2
\]

Here, the type noalias classifies finite maps from pointers of type \text{ptr} to values, \(h_1 \bullet h_2\) is the disjoint union of \(h_1\) and \(h_2\), and \(x \mapsto v\) is a singleton heap consisting solely of the pointer \(x\), storing the value \(v\). The disjoint union may be undefined if \(h_1\) and \(h_2\) overlap, so we need a predicate \text{def} \(h\), declaring that \(h\) is not undefined. Consequently, \text{def} \((h_1 \bullet h_2)\) holds iff \(h_1\) and \(h_2\) are disjoint heaps. Finally, the conclusion \(x_1 \neq x_2\) is in fact a term of type \text{bool}, which Coq implicitly coerces to the proposition \((x_1 \neq x_2) = \text{true}\). The noalias lemma states that \(x_1\) and \(x_2\) are not aliased, if they are known to belong to disjoint singleton heaps.

Now suppose we want to prove a goal consisting of a number of no-aliasing facts, e.g.,

\[(x_1 \neq x_2) \&\& (x_2 \neq x_3) \&\& (x_3 \neq x_1),\]

under the following hypothesis:

\[D : \text{def } (i_1 \cdot (x_1 \mapsto v_1 \cdot x_2 \mapsto v_2) \bullet (i_2 \cdot x_3 \mapsto v_3))\]

Before noalias can be applied to prove, say, \(x_2 \neq x_3\), the disjoint union in \(D\) will have to be rearranged, so that the pointers \(x_2\) and \(x_3\) appear at the top of the union, as in:

\[D' : \text{def } (x_2 \mapsto v_2 \cdot x_3 \mapsto v_3 \cdot i_1 \cdot i_2 \cdot x_1 \mapsto v_1)\]

Otherwise, the noalias lemma will not apply. Because \(\cdot\) is commutative and associative, the rearrangement is sound, but it is tedious to perform by hand, and it is not very robust under adaptation. Indeed, if the user goes back and changes the input heap in \(D\), a new rearrangement is necessary. Furthermore, the tedium is exacerbated by the need for different rearrangements in proving \(x_1 \neq x_2\) and \(x_3 \neq x_1\).

The most effective solution would be for the type checker to somehow automatically recognize that the heap expression from \(D\) is in fact equivalent to some form required by noalias. Unfortunately, none of the proof assistants that we are aware of provide such automatic reasoning primitively. Instead, they typically provide a separate language for writing tactics, which are customized procedures for solving a class of proof obligations. For example, one can write an \text{auto_noalias} tactic to solve a goal like \(x_2 \neq x_3\) by automatically converting the assumption \(D\) into \(D'\) and then applying the noalias lemma. However, while tactics have been deployed successfully (and with impressive dexterity) in a variety of scenarios (Chlipala, 2008; Chlipala, 2011), they are beset by certain fundamental limitations.

The primary drawback of tactics is that they lack the precise typing of the theorem prover’s base logic (and in the case of Coq, they are essentially untyped). This can make
them much more difficult to maintain than lemmas, as changes in basic definitions do not necessarily raise type errors in the code of the tactics affected by the changes. Rather, type checking is performed on the goals obtained during tactic execution, resulting in potentially obscure error messages and unpredictable proof states in the case of failure. Moreover, the behavior of a tactic typically cannot be specified, nor can it be verified against a specification.

Due to their lack of precise typing, tactics suffer a second-class status, in the sense that they may not be used as flexibly as lemmas. For example, suppose the pointer (in-)equalities we want to resolve are embedded in a larger context, e.g.,

\[ G: \text{if } (x_2 == x_3) \&\& (x_1 != x_2) \text{ then } E_1 \text{ else } E_2 \]

In this situation, we cannot apply the `auto_noalias` tactic directly to reduce \((x_2 == x_3)\) and \((x_1 != x_2)\) to false and true, respectively, since those (in-)equalities are not the top-level goal. Coq’s `rewrite` primitive is designed precisely for this situation—it enables one to reduce all (in-)equalities within \(G\) that match the conclusion of a particular lemma—but it is not applicable to tactics (like `auto_noalias`).

Thus, with the `auto_noalias` tactic, we are left with essentially two options: (1) use it to prove a bespoke lemma about one specific inequality (say, \(x_1 != x_2\)), perform a rewrite using that lemma, and repeat for other (in-)equalities of interest, or (2) implement another custom tactic that crawls over the goal \(G\) searching for any and all (in-)equalities that `auto_noalias` might resolve. The former option sacrifices the benefits of automation, while the latter option redundantly duplicates the functionality of `rewrite`.

Ideally, we would prefer instead to have a way of writing `auto_noalias` as a lemma rather than a tactic. Had we such a lemma, we could give it a precisely typed specification, we could `rewrite` the goal \(G\) with it directly, and we could also compose it with other lemmas. For instance, we could use ordinary function composition to compose it with the standard lemma

\[ \text{negbTE} : \forall b:\text{bool}. \neg b = \text{true} \rightarrow b = \text{false}, \]

thus transforming `auto_noalias` into a rewrite rule for positive facts of the form \((x_2 == x_3) = \text{false}\). Consequently, we could apply `rewrite` \((\text{negbTE} (\text{auto_noalias} D))\) to the goal \(G\), thereby reducing it to \(E_2\).

The question of how to support automation, while remaining within the first-class world of lemmas, is the subject of this paper.

### 1.1 Contributions

We propose a novel and powerful approach to proof automation in Coq, which avoids the aforementioned problems with tactics by allowing one to program custom automated routines within the expressive dependent type system of Coq itself. In particular, we will be able to rephrase the `noalias` lemma so that it can automatically analyze its heap-definedness hypothesis \(D\) in order to derive whatever valid pointer inequalities are needed, without any manual intervention from the user. Our proposal is much more general, however, and we will illustrate it on a variety of different and significantly more involved examples than just `noalias`.

https://doi.org/10.1017/S0956796813000051 Published online by Cambridge University Press
Our approach involves a sophisticated application of Coq’s canonical structures, which have existed in Coq for quite some time (Saïbi, 1997), but with sparse documentation and (perhaps as a consequence) relatively little use. At a high level, canonical structures may be viewed as a generalization of Haskell’s type classes (Wadler & Blott, 1989; Hall et al., 1996), in the sense that they provide a principled way to construct default dictionaries of values and methods—and hence support overloading and implicit program construction—as part of the type inference process.

However, unlike in Haskell, where the construction of canonical instances is keyed solely on the type belonging to a certain type class, instance construction in Coq may be keyed on terms as well. This, together with Coq’s support for backtracking during type inference, enables a very flexible style of dependently-typed logic programming. Furthermore, since canonical structures can embed proofs of interesting invariants about the instance fields being computed, one can use them to implement custom algorithms (in logic-programming style) together with proofs of their (partial) correctness. Thus, just as Haskell type classes are used to infer the canonical implementation of an overloaded term at a given type, canonical structures can be used to infer the canonical proof of an overloaded lemma for a given instantiation of its parameters. We feel this constitutes a beautiful application of the Curry–Howard correspondence between proofs and programs in Coq.

Intuitively, our approach works as follows. Suppose we want to write a lemma whose application may need to trigger an automated solution to some subproblem (e.g., in the case of noalias, the problem of testing whether two pointers x₁ and x₂ appear in disjoint subheaps of the heap characterized by the heap-definedness hypothesis D). In this case, we define a structure (like a type class) to encapsulate the problem whose solution we wish to automate, and we encode the algorithm for solving that problem—along with its proof of correctness—in the canonical instances of the structure. Then, when the lemma is applied to a particular goal, unification of the goal with the conclusion of the lemma will trigger the construction of a canonical instance of our structure that solves the automation problem for that goal. For example, if auto_noalias is the overloaded version of noalias, and we try to apply (auto_noalias D) to the goal of proving x₂ ≠ x₃, type inference will trigger construction of a canonical instance proving that the heap characterized by D contains bindings for x₂ and x₃ in two disjoint subheaps. (This is analogous to how the application of an overloaded function in Haskell triggers the construction of a canonical dictionary that solves the appropriate instantiation of its type class constraints.) Although we have described the approach here in terms of backward reasoning, one may also apply overloaded lemmas using forward reasoning, as we will see in Section 5.

Key to the success of our whole approach is the Coq type inference engine’s use of syntactic pattern-matching in determining which canonical instances to apply when solving automation problems. Distinguishing between syntactically distinct (yet semantically equivalent) terms and types is essential if one wishes to simulate the automation power of

---

1 It is folklore that one can simulate logic programming to some extent using Haskell’s multi-parameter classes with functional dependencies (Jones, 2000) or with associated types (Chakravarty et al., 2005), but Haskell’s lack of support for backtracking during type inference limits what kinds of logic programming idioms are possible.
tactics. However, it is also an aspect of our approach that Coq’s type system cannot account for because it does not observe such syntactic distinctions. Fortunately, our reliance on Coq’s unification algorithm for analysis of syntax is the only aspect of our approach that resides outside of Coq’s type system, unlike tactics, which are wholly extra-linguistic.

Perhaps the greatest challenge in making our approach fly is in developing effective and reliable ways of circumventing certain inherent restrictions of Coq’s canonical structures, which were not designed with our ambitious application in mind. In particular, in order to implement various useful forms of automation using canonical structures, it is critically important to be able to write overlapping instances, but also to control the order in which they are considered and the order in which unification subproblems are solved. None of these features are supported primitively, but they are encodable using a series of simple “design patterns”, which form the core technical contribution of this paper.

We illustrate these patterns through several realistic examples involving reasoning about heaps, pointers and aliasing. All of these examples have been implemented and tested in the context of Hoare Type Theory (HTT) (Nanevski et al., 2010), where they have replaced often-used tactics. The code in this paper and HTT itself is built on top of Ssreflect (Gonthier & Mahboubi, 2010), which is a recent extension of Coq providing a very robust language of proofs, as well as libraries for reflection-based reasoning. However, in the current paper, we assume no prior knowledge of Coq, Ssreflect, or canonical structures themselves. We will remain close, but not adhere strictly, to the Coq notation and syntax. All of our sources are available on the web (Gonthier et al., 2012).

The rest of the paper is structured as follows. In Section 2, we review the basic ideas behind canonical structures. We show how they can be used for function overloading and, more generally, logic programming, including a design pattern for imposing ordering on overlapping clauses of a logic program. We utilize such logic programming in order to implement the automation procedures at the heart of all our overloaded lemmas.

In subsequent sections, we proceed to illustrate a number of ways in which lemmas can be overloaded so that they automate various proof tasks related to heaps, pointers and aliasing. Section 3 presents our first overloaded lemma, which automates a simple task of checking if a pointer appears in the domain of a heap. We walk through this relatively simple warmup example in full gory detail, so that the reader may understand exactly how we harness the Coq unification algorithm for our purposes. Section 4 uses overloading to implement a much more sophisticated example, namely a reflective procedure for cancelling common terms from heap equations. Section 5 illustrates a pattern that relies on higher-order functions to implement an automated procedure for symbolic evaluation in separation logic. Section 6 explores the problem of how to effectively compose two overloaded lemmas so that the result may be applied in both forward and backward reasoning, leading eventually to a final pattern for reordering unification subproblems. Lastly, Section 7 discusses related work and Section 8 concludes.

2 Basics of canonical structures

In this section, we provide a quick introduction to the basics of canonical structure programming, leading up to our first important “design pattern”—tagging—which is critical for supporting ordering of overlapping instance declarations.
2.1 “Type class” programming

In the literature and everyday use of Coq, the word “structure” is used interchangeably (and confusingly) to mean both dependent records and the types they inhabit. To disambiguate, in this paper we use structure for the type, instance for the value, and canonical instance for a canonical value of a certain type. We will use the term canonical structures only when referring generally to the use of all of these mechanisms in tandem.

The following definition is a simplified example of a structure (i.e., type) taken from the standard Ssreflect library (Gonthier & Mahboubi, 2010):

```coq
structure eqType := EqType { sort : Type;
  equal : sort → sort → bool;
  _ : ∀ x y : sort. equal x y ↔ x = y }
```

The definition makes eqType a record type, with EqType as its constructor, taking three arguments: a type sort, a boolean binary operation equal on sort, and a proof that equal decides the equality on sort. For example, one possible eqType instance for the type bool, may be

```coq
eqType_bool := EqType bool eq_bool pf_bool
```

where eq_bool x y := (x && y) || (!x && !y), and pf_bool is a proof, omitted here, that ∀x y : bool. eq_bool x y ↔ x = y. (Note that, in Coq, although it may seem as though EqType is declared as taking a single record argument with three components, applications of EqType pass the three arguments in curried style.)

The labels for the record fields serve as projections out of the record, so the definition of eqType also introduces the constants:

```coq
  sort : eqType → Type
  equal : ∀T:eqType. sort T → sort T → bool
```

We do not care to project out the proof component of the record, so we declare it anonymous by naming it with an underscore.

**Notational Convention 1**

We will usually omit the argument T of equal, and write equal x y instead of equal T x y, as T can be inferred from the types of x and y. We use the same convention for other functions as well, and make implicit such arguments that can be inferred from the types of other arguments. This is a standard notational convention in Coq.

It is also very useful to define generic instances. For example, consider the eqType instance for the pair type A × B, where A and B are themselves instances of eqType:

```coq
eqType_pair (A B : eqType) :=
  EqType (sort A × sort B) (eq_pair A B) (pf_pair A B)
```

where

```coq
  eq_pair (A B : eqType) (u v : sort A × sort B) :=
  equal (π₁ u) (π₁ v) && equal (π₂ u) (π₂ v)
```

and pf_pair is a proof, omitted just like pf_bool above, that ∀A B : eqType. ∀x y : (sort A × sort B). eq_pair x y ↔ x = y.
Declaring both eqType_bool and eqType_pair now as canonical instances—using Coq’s canonical keyword—will have the following effect: whenever the type checker is asked to type a term like equal \((b_1, b_2) (c_1, c_2)\), where \(b_1, b_2, c_1\) and \(c_2\) are of type bool, it will generate a unification problem matching the expected and inferred type of the first non-implicit argument of equal, that is,

\[
\text{sort } \ ?T \cong \text{bool} \times \text{bool}
\]

for some unification variable \(?T\), generated implicitly at the application of equal. It will then try to solve this problem using the canonical instance eqType_pair, resulting in two new unification subproblems, for fresh unification variables \(?A\) and \(?B\):

\[
\text{sort } \ ?A \cong \text{bool} \quad \text{sort } \ ?B \cong \text{bool}
\]

Next, it will choose \(?A \cong \text{eqType} \text{bool}\) and \(?B \cong \text{eqType} \text{bool}\), with the final result that equal \((b_1, b_2) (c_1, c_2)\) reduces implicitly to eq.bool \(b_1 c_1 \& \& \text{eq.bool} b_2 c_2\), as one would expect.

In this manner, canonical instances can be used for overloading, similar to the way type classes are used in Haskell (Wadler & Blott, 1989; Hall et al., 1996). We can declare a number of canonical eqType instances, for various primitive types, as well as generic instances for type constructors (like the pair example above). Then we can uniformly write equal \(x y\), and the type checker will compute the canonical implementation of equality at the types of \(x\) and \(y\) by solving for equal’s implicit argument \(T\).

Generalizing from eqType to arbitrary structures \(S\), the declaration of an instance \(V : S\) as canonical instructs the type checker that for each projection proj of the structure \(S\), and \(c\) the head symbol of proj \(V\), the unknown \(X\) in the unification equation

\[
\text{proj } X \cong c \ x_1 \ldots x_n \quad (*)
\]

should by default be solved by unifying \(X \cong V\). For instance, in the unification equation generated by the equal application above, the projector proj is sort, the head constant \(c\) is \((\cdot \times \cdot)\), and the head constant arguments \(x_1 \ldots x_n\) are bool and bool. On the other hand, if the head symbol of proj \(V\) is a variable, then \(V\) is chosen for unification with \(X\) irrespectively of \(c\) in equation \((*)\). In the latter case, we refer to \(V\) as a default canonical instance for proj.

We emphasize that: (1) to control the number of \((\text{proj}, c)\)-pairs that the type checker has to remember, we will frequently anonymize the projections if they are not important for the application, as in the case of the proof component in eqType above; (2) there can only be one specified canonical instance for any given proj and \(c\). In particular, overlapping canonical instances for the same proj and \(c\) are not permitted. As we will see shortly, however, there is a simple design pattern that will allow us to circumvent this limitation.

### 2.2 “Logic” programming

Although the eqType example is typical of how canonical structures are used in much existing Coq code, it is not actually representative of the style of canonical structure

---

2 It is worth noting that Coq also provides a built-in type class mechanism, but this feature is independent of canonical structures. We discuss Coq type classes more in Section 7.
programming that we explore in this paper. Our idiomatic style is closer in flavor to logic programming and relies on the fact that, unlike in Haskell, the construction of canonical instances in Coq can be guided not only by the structure of types (such as the sort projection of eqType) but by the structure of terms as well.

To make matters concrete, let us consider a simple automation task, one which we will employ gainfully in Section 3 when we present our first “overloaded lemma”. We will first present a naïve approach to solving the task, which almost works; the manner in which it fails will motivate our first “design pattern” (Section 2.3).

The task is as follows: search for a pointer x in the domain of a heap h. If the search is successful, that is, if h is of the form

\[ \cdots \cdot (\cdots \cdot x \mapsto v \cdot \cdots) \cdot \cdots, \]

then return a proof that \( x \in \text{dom } h \). To solve this task using canonical structures, we will first define a structure find:

```coq
structure find x := Find { heap_of : heap; 
_ : spec x heap_of }
```

where spec is defined as

\[ \text{spec } x h := \text{def } h \rightarrow x \in \text{dom } h \]

The first thing to note here is that the structure find is parameterized by the pointer x (causing the constructor Find to be implicitly parameterized by x as well). This is a common idiom in canonical structure programming—and we will see that structure parameters can be used for various different purposes—but here x may be viewed simply as an “input” to the automation task. The second thing to note here is that the structure has no type component, only a heap_of projection, together with a proof that \( x \in \text{dom heap_of} \) (under the assumption that heap_of is well-defined).

The search task will commence when some input heap h gets unified with heap_of X for an unknown X : find x, at which point Coq’s unification algorithm will recursively deconstruct h in order to search for a canonical implementation of X such that heap_of X = h. If that search is successful, the last field of X will be a proof of spec x h, which we can apply to a proof of def h to obtain a proof of \( x \in \text{dom } h \), as desired. (By way of analogy, this is similar to what we previously did for eqType_pair. The construction of a canonical equality operator at a given type A will commence when A is unified with sort T for an unknown T : eqType, and the unification algorithm will proceed to solve for T by recursively deconstructing A and composing the relevant canonical instances.)

The structure find provides a formal specification of what a successful completion of the search task will produce, but now we need to actually implement the search. We do that by defining several canonical instances of find corresponding to the different cases of the
recursive search, and relying on Coq’s unification algorithm to perform the recursion:

\[
\text{canonical} \quad \text{found} \quad \text{struct} \quad A \quad x \quad (v : A) := \quad \text{Find} \quad x \quad (x \mapsto v) \quad (\text{found} \_ \text{pf} \ A \ x \ v)
\]

\[
\text{canonical} \quad \text{left} \quad \text{struct} \quad x \quad h \quad (f : \text{find} \ x) := \quad \text{Find} \quad x \quad ((\text{heap} \_ \text{of} \ f) \cdot h) \quad (\text{left} \_ \text{pf} \ x \ h \ f)
\]

\[
\text{canonical} \quad \text{right} \quad \text{struct} \quad x \quad h \quad (f : \text{find} \ x) := \quad \text{Find} \quad x \quad (h \cdot (\text{heap} \_ \text{of} \ f)) \quad (\text{right} \_ \text{pf} \ x \ h \ f)
\]

Note that the first argument to the constructor \text{Find} in these instances is the parameter \(x\) of the find structure.

The first instance, \text{found} \_ \text{struct}, corresponds to the case where the heap \_ \text{of} projection is a singleton heap whose domain contains precisely the \(x\) we’re searching for. (If the heap is \(y \mapsto v\) for \(y \neq x\), then unification fails.) The second and third instances, \text{left} \_ \text{struct} and \text{right} \_ \text{struct}, handle the cases where the heap \_ \text{of} projection is of the form \(h_1 \cdot h_2\), and \(x\) is in the domain of \(h_1\) or \(h_2\), respectively. Note that the recursive nature of the search is implicit in the fact that the latter two instances are parameterized by instances \(f : \text{find} \ x\) whose heap \_ \text{of} projections are unified with the subheaps \(h_1\) or \(h_2\) of the original heap \_ \text{of} projection.

\textit{Notational Convention 2}

In the declarations above, \text{found} \_ \text{pf}, \text{left} \_ \text{pf} and \text{right} \_ \text{pf} are proofs, witnessing that \text{spec} relates \(x\) and the appropriate heap expression. We omit the proofs here, but they are available in our source files. From now on, we omit writing such explicit proofs in instances, and simply replace them with “...”, as in: \text{Find} \quad x \quad ((\text{heap} \_ \text{of} \ f) \cdot h) \ldots

Unfortunately, this set of canonical instances does not quite work. The trouble is that \text{left} \_ \text{struct} and \text{right} \_ \text{struct} are overlapping instances since both match against the same head symbol (namely, \(\cdot\)), and overlapping instances are not permitted in Coq. Moreover, even if overlapping instances were permitted, we would still need some way to tell Coq that it should try one instance first and then, if that fails, to backtrack and try another. Consequently, we need some way to deterministically specify the order in which overlapping instances are to be considered. For this, we introduce our first design pattern.

\textit{2.3 Tagging: a technique for ordering canonical instances}

Our approach to ordering canonical instances is, in programming terms, remarkably simple. However, understanding why it actually works is quite tricky because its success relies critically on an aspect of Coq’s unification algorithm that (a) is not well known, and (b) diverges significantly from how unification works in, say, Haskell. We will thus first illustrate the pattern concretely in terms of our \text{find} example, and then explain afterwards how it solves the problem.

\textbf{The pattern.} First, we define a “tagged” version of the type of thing we are recursively analyzing—in this case, the heap type:

\[
\text{structure} \quad \text{tagged} \_ \text{heap} := \quad \text{Tag} \quad \{ \text{untag} : \text{heap} \}
\]
This structure declaration also introduces two functions witnessing the isomorphism between heap and tagged_heap:

\[
\begin{align*}
\text{Tag} & : \text{heap} \rightarrow \text{tagged_heap} \\
\text{untag} & : \text{tagged_heap} \rightarrow \text{heap}
\end{align*}
\]

Then, we modify the find structure to carry a tagged_heap instead of a plain heap, i.e., we declare

\[
\text{spec } x (h : \text{tagged_heap}) := \\
\text{def } (\text{untag } h) \rightarrow x \in \text{dom } (\text{untag } h)
\]

\[
\text{structure } \text{find } x := \text{Find } \{ \text{heap_of } : \text{tagged_heap}; \\
\quad : \text{spec } x \text{ heap_of} \}
\]

Next, we define a sequence of synonyms for Tag, one for each canonical instance of find. Importantly, we define the tag synonyms in the reverse order in which we want the canonical instances to be considered during unification, and we make the last tag synonym in the sequence be the canonical instance of the tagged_heap structure itself. (The order does not matter much in this particular example, but it does in other examples in the paper.)

\[
\begin{align*}
\text{right_tag } h & := \text{Tag } h \\
\text{left_tag } h & := \text{right_tag } h \\
\text{canonical found_tag } h & := \text{left_tag } h
\end{align*}
\]

Notice that found_tag is a default instance for the untag projector matching any heap \( h \) (cf. the end of Section 2.1).

Finally, we modify each canonical instance so that its heap_of projection is wrapped with the corresponding tag synonym.

\[
\begin{align*}
\text{canonical found_struct } A x (v : A) := \\
\text{Find } x (\text{found_tag } (x \mapsto v)) \ldots
\end{align*}
\]

\[
\begin{align*}
\text{canonical left_struct } x h (f : \text{find } x) := \\
\text{Find } x (\text{left_tag } ((\text{untag } (\text{heap_of } f)) \bullet h)) \ldots
\end{align*}
\]

\[
\begin{align*}
\text{canonical right_struct } x h (f : \text{find } x) := \\
\text{Find } x (\text{right_tag } (h \bullet (\text{untag } (\text{heap_of } f)))) \ldots
\end{align*}
\]

The explanation. The key to the tagging pattern is that, by employing different tags for each of the canonical instance declarations, we are able to syntactically differentiate the head constants of the heap_of projections, thereby circumventing the need for overlapping instances. But the reader is probably wondering: (1) how can semantically equivalent tag synonyms differentiate anything? and (2) what’s the deal with defining them in the reverse order?

The answer to (1) is that Coq does not unfold all definitions automatically during the unification process—it only unfolds the definition of a term like found_tag \( h \) automatically if that term is unified with something else and the unification fails (see the next paragraph). This stands in contrast to Haskell type inference, which implicitly expands all (type) synonyms right away. Thus, even though found_tag, left_tag, and right_tag are all semantically

https://doi.org/10.1017/S0956796813000051 Published online by Cambridge University Press
equivalent to Tag, the unification algorithm can distinguish between them, rendering the
three canonical instances of find non-overlapping.

The answer to (2) is as follows. By making the last tag synonym found_tag the sole
canonical instance of tagged_heap, we guarantee that unification always pattern-matches
against the found_struct case of the search algorithm first before any other. To see this,
observe that the execution of the search for \( x \) in \( h \) will get triggered when a unification
problem arises of the form

\[
\text{untag } (\text{heap_of } ?f) \equiv h,
\]

for some unknown \( ?f : \text{find} \). Since found_tag is a default canonical instance, the problem
will be reduced to unifying

\[
\text{heap_of } ?f \equiv \text{found_tag } h
\]

As found_struct is the only canonical instance of find whose heap_of projection has
found_tag as its head constant, Coq will first attempt to unify \( ?f \) with some instantiation
of found_struct. If \( h \) is a singleton heap containing \( x \), then the unification will succeed.
Otherwise, Coq will backtrack and try unfolding the definition of found_tag \( h \) instead,
resulting in the new unification problem

\[
\text{heap_of } ?f \equiv \text{left_tag } h,
\]

which will in turn cause Coq to try unifying \( ?f \) with some instantiation of left_struct. If
that fails again, left_tag \( h \) will be unfolded to right_tag \( h \) and Coq will try right_struct. If
in the end that fails as well, then it means that the search has failed to find \( x \) in \( h \), and Coq
will correctly flag the original unification problem as unsolvable.

### 3 A simple overloaded lemma

Let us now attempt our first example of lemma overloading, which makes immediate use
of the find structure that we developed in the previous section. First, here is the non-
overloaded version:

\[
\text{indom} : \forall A: \text{Type}. \forall x: \text{ptr}. \forall v: A. \forall h: \text{heap}.
\text{def } (x \mapsto v \bullet h) \rightarrow x \in \text{dom } (x \mapsto v \bullet h)
\]

The indom lemma is somewhat simpler than noalias from Section 1, but the problems in
applying them are the same—neither lemma is applicable unless its heap expressions are
of a special syntactic form, with the relevant pointer(s) at the top of the heap.

To lift this restriction, we will rephrase the lemma into the following form:

\[
\text{indomR} : \forall x: \text{ptr}. \forall f: \text{find} x.
\text{def } (\text{untag } (\text{heap_of } f)) \rightarrow
x \in \text{dom } (\text{untag } (\text{heap_of } f))
\]

The lemma is now parameterized over an instance \( f \) of structure \( \text{find} x \), which we know—
just from the definition of \( \text{find} \) alone—contains within it a heap \( h = \text{untag } (\text{heap_of } f) \),
together with a proof of \( \text{def } h \rightarrow x \in \text{dom } h \). Based on this, it should come as no surprise
that the proof of \( \text{indomR} \) is trivial (it is a half-line long in Ssreflect). In fact, the lemma
is really just the projection function corresponding to the unnamed spec component from
the find structure, much as the overloaded equal function from Section 2.1 is a projection function from the eqType structure.

3.1 Applying the lemma: the high-level picture

To demonstrate the automated nature of indomR on a concrete example, we will explain how Coq type inference proceeds when indomR is applied to prove the goal

\[ z \in \text{dom } h \]

in a context where \( x \ y \ z : \text{ptr}, u \ v \ w : A \) for some type \( A \), \( h : \text{heap} := x \mapsto u \cdot y \mapsto v \cdot z \mapsto w \), and \( D : \text{def } h \). For the moment, we will omit certain details for the sake of clarity; in the next subsection, we give a much more detailed explanation.

To begin with, let us first explain the steps involved in the application of a lemma to some goal, and how this produces the equation needed to solve the instance \( f \) of the structure.

When a lemma (e.g., indomR) is applied to a goal, the following process takes place:

1. The lemma’s formal parameters are turned into unification variables \(?x\) and \(?f\):
   
   find \(?x\), which will be subsequently constrained by the unification process. (Hereafter, we will use \(?\) to denote unification variables, with previously unused \(?x\)’s denoting “fresh” unification variables.)

2. The lemma’s conclusion \(?x \in \text{dom (untag (heap_of } ?f))\) is unified with the goal.

Given this last step, the system tries to unify

\[ ?x \in \text{dom (untag (heap_of } ?f)) \cong z \in \text{dom } h \]

solving subproblems from left to right, that is, first getting \(?x = z\), and then

\[ \text{untag (heap_of } ?f) \cong h \]

By canonicity of found_tag, it then tries to solve

\[ \text{heap_of } ?f \cong \text{found_tag } h \]

Expanding the heap variable, and since \( \cdot \) is left-associative, this is equivalent to

\[ \text{heap_of } ?f \cong \text{found_tag } ((x \mapsto u \cdot y \mapsto v) \cdot z \mapsto w) \]

At this point, guided by the instances we defined in the previous section, the search for a canonical instantiation of \(?f\) begins. Coq will first try to instantiate \(?f\) with found_struct, but this attempt will fail when it tries to unify the entire heap with a singleton heap. Then, Coq will try instantiating \(?f\) instead with left_struct, which leads it to create a fresh variable \(?f_2 : \text{find } z\) and recursively solve the equation untag (heap_of \(?f_2\)) \cong x \mapsto u \cdot y \mapsto v. This attempt will ultimately fail again, because \( z \) is not in \( x \mapsto u \cdot y \mapsto v\), the left subheap of the original \( h \). Finally, Coq will backtrack and try to instantiate \(?f\) with right_struct, which will lead it to create a fresh variable \(?f_3 : \text{find } z\) and recursively solve the equation untag (heap_of \(?f_3\)) \cong z \mapsto w. This final attempt will indeed succeed by instantiating \(?f_3\) with found_struct, since the right subheap of \( h \) is precisely the singleton heap we are looking for.
Putting the pieces together, the unification algorithm instantiates \(?f\) with
\[
?f = \text{right\_struct } z (x \mapsto u \cdot y \mapsto v) (\text{found\_struct } z \ w)
\]
Effectively, the heap\_of component of \(?f\) contains the (tagged) heap \(h\) that was input to the search, and the proof component contains the output proof that \(z\) is in the domain of \(h\).

### 3.2 The gory details

To understand better how it works, we will now spell out the “trace” of how Coq’s unification algorithm implements proof search in the example above, with a particular emphasis on how it treats resolution of canonical instances. This knowledge is not critical for understanding most of the examples in the paper—indeed, the whole point of our “design patterns” is to avoid the need for one to think about unification at this level of gory detail. But it will nonetheless be useful in understanding why the design patterns work, as well as how to control Coq’s unification algorithm in more complex examples where the design patterns do not immediately apply.

Let us look again at the initial equation that started the search:
\[
\text{untag } (\text{heap\_of } ?f) \equiv h
\]
As mentioned before, the canonicity of \(\text{found\_tag}\) reduces this to solving
\[
\text{heap\_of } ?f \equiv \text{found\_tag } h
\]
Unification tries to instantiate \(?f\) with \(\text{found\_struct}\), but for that it must unify the entire heap \(h\) with \(z \mapsto ?v\), which fails. Before giving up, the system realizes it can unfold the definitions of \(h\) and of \(\text{found\_tag}\), yielding
\[
\text{heap\_of } ?f \equiv \text{left\_tag } ((x \mapsto u \cdot y \mapsto v) \cdot z \mapsto w)
\] (1)
With this unfolding, \(?f\) has become eligible for instantiation by \(\text{left\_struct}\), since \(\text{left\_struct}\) is the canonical instance of the find structure with head constant \(\text{left\_tag}\). To figure out if/whether this instantiation is possible, Coq will engage in the following procedure, which we will describe here quite carefully—and in as general a manner as possible—since it is important for understanding subsequent, more complex examples.

To instantiate \(?f\) with \(\text{left\_struct}\), Coq first “opens” the right-hand side of the definition of \(\text{left\_struct}\) by generating fresh unification variables for each of \(\text{left\_struct}\)’s formal parameters: \(?y : \text{ptr}, \ ?h : \text{heap}, \ \text{and } ?f_2 : \text{find } ?y\). It will eventually unify \(?f\) with \(\text{left\_struct } ?y \ ?h \ ?f_2\), but before doing that, it must figure out how to marry together two sources of (hopefully compatible) information about \(?f\): the unification goal (\(i.e.,\) Equation (1)) and the definition of \(\text{left\_struct}\). Each of these provides information about:

1. The type of the structure we are solving for (in this case, \(?f\)). From the initial unification steps described in Section 3.1, we already know that \(?f\) must have type \(\text{find } z\), while the type of \(\text{left\_struct } ?y \ ?h \ ?f_2\) is \(\text{find } ?y\). This leads to the unification
\[
?y \equiv z
\]
2. The (full) value of the projection in question (in this case, \(\text{heap\_of } ?f\)). From Equation (1), we know that \(\text{heap\_of } ?f\) must equal \(\text{left\_tag } ((x \mapsto u \cdot y \mapsto v) \cdot z \mapsto w)\),
while from the definition of left_struct, we know that heap_of (left_struct ?y ?h ?f₂) equals left_tag ((untag (heap_of ?f₂)) • ?h). This leads to the unification

\[
\text{left_tag } ((\text{untag } (\text{heap_of } ?f₂)) • ?h) \equiv \text{left_tag } ((x \mapsto u \cdot y \mapsto v) • z \mapsto w),
\]

which in turn induces the following unification equations:

\[
\begin{align*}
\text{untag } (\text{heap_of } ?f₂) & \equiv x \mapsto u \cdot y \mapsto v \\
?h & \equiv z \mapsto w \\
?f & \equiv \text{left_struct } ?y ?h ?f₂
\end{align*}
\]

Putting it all together, the attempt to instantiate \( ?f \) with left_struct generates the following unification problems, which Coq processes in order:

\[
\begin{align*}
?y & \equiv z \\
\text{untag } (\text{heap_of } ?f₂) & \equiv x \mapsto u \cdot y \mapsto v \\
?h & \equiv z \mapsto w \\
?f & \equiv \text{left_struct } ?y ?h ?f₂
\end{align*}
\]

Note here that the order matters! For instance, the first equation will be resolved immediately, thus concretizing the type of \( ?f₂ \) to find \( z \). It is important that this happens before solving the second equation, so that when we attempt to solve the second equation we know what pointer (\( z \)) we are searching for in the heap \( x \mapsto u \cdot y \mapsto v \). (Otherwise, we would be searching for the unification variable \( ?y \), which would produce senseless results.)

Attempting to solve the second equation, Coq again applies found_tag and found_struct and fails. Then, it unfolds found_tag to get left_tag and the following equation:

\[
\text{heap_of } ?f₂ \equiv \text{left_tag } (x \mapsto u \cdot y \mapsto v)
\]

It attempts to instantiate \( ?f₂ \) with left_struct, by the same procedure as described above, obtaining the following equations:

\[
\begin{align*}
?y' & \equiv z \\
\text{untag } (\text{heap_of } ?f₃) & \equiv x \mapsto u \quad \text{where } ?f₃ : \text{find } ?y'
\end{align*}
\]

After solving the first one, the attempt to solve the second one will fail. Specifically, Coq will first try to use found_struct as before; however, this will not work because, although the heap in question \( (x \mapsto u) \) is a singleton heap, the pointer in its domain is not \( z \). Unfolding to left_struct and right_struct will not help because those solutions only apply to heaps with \( \cdot \) as the top-level constructor.

Rolling back to Equation (2), Coq unfolds left_tag to right_tag and tries to instantiate \( ?f₂ \) with right_struct. As before, it fails because \( z \) is not \( y \).

Rolling back further to Equation (1), Coq unfolds left_tag to right_tag, resulting in

\[
\text{heap_of } ?f \equiv \text{right_tag } ((x \mapsto u \cdot y \mapsto v) • z \mapsto w).
\]
It then chooses right_struct and applies the same procedure as before to obtain the following equations.

\[
\begin{align*}
?x' & \equiv z \\
?h' & \equiv x \mapsto u \cdot y \mapsto v \\
\text{untag (heap_of } ?f'_2) & \equiv z \mapsto w \\
?f & \equiv \text{right_struct } ?x' \ ?h' \ ?f'_2
\end{align*}
\]

The first two equations unify immediately, after which the third one is solved by applying found_tag and choosing found_struct for \(?f'_2\). After that, the third equation also unifies right away, producing the final result

\[?f = \text{right_struct } z (x \mapsto u \cdot y \mapsto v) (\text{found_struct } z \ w)\]

### 4 Reflection: turning semantics into syntax

As canonical structures are closely coupled with the type checker, it is possible to fruitfully combine the logic-programming idiom afforded by canonical structures together with ordinary functional programming in Coq. In this section, we illustrate the combination by developing a thoroughly worked example of an overloaded lemma for performing cancellation on heap equations. In our implementation of Hoare Type Theory, this lemma is designed to replace an often used, but rather complicated and brittle, tactic.

Mathematically, cancellation merely involves removing common terms from disjoint unions on the two sides of a heap equation. For example, if we are given an equation

\[x \mapsto v_1 \cdot (h_3 \cdot h_4) = h_4 \cdot x \mapsto v_2\]

and we know that the subheaps are disjoint (i.e., the unions are defined), then we can extract the implied equations

\[v_1 = v_2 \land h_3 = \text{empty}\]

We will implement the lemma in two stages. The first stage is a canonical structure program, which reflects the equation, that is, turns the equation on heap expressions into an abstract syntax tree (or abstract syntax list, as it will turn out). Then the second stage is a functional program, which cancels common terms from the syntax tree. Notice that the functional program from the second stage cannot work directly on the heap equation for two reasons: (1) it needs to compare heap and pointer variable names, and (2) it needs to pattern-match on function names, since in HTT heaps are really partial maps from locations to values, and \(\mapsto\) and \(\cdot\) are merely convenient functions for constructing them. As neither of these is possible within Coq’s base logic, the equation has to be reflected into syntax in the first stage. The main challenge then is in implementing reflection, so that the various occurrences of one and the same heap variable or pointer variable in the equation are ascribed the same syntactic representation.
4.1 Cancellation

Since the second stage is simpler, we explain it first. For the purposes of presentation, we begin by restricting our pointers to only store values of some predetermined type $T$ (although in Section 4.3 we will generalize it to any type, as in our actual implementation). The data type that we use for syntactic representation of heap expressions is the following:

$$\text{elem} := \text{Var of nat} | \text{Pts of nat} \& T$$

$$\text{term} := \text{seq elem}$$

An element of type `elem` identifies a heap component as being either a heap variable or a points-to clause $x \mapsto v$. In the first case, the component is represented as `Var n`, where $n$ is an index identifying the heap variable in some environment (to be explained below). In the second case, the component is represented as `Pts m v`, where $m$ is an index identifying the pointer variable in an environment. We do not perform any reflection on $v$, as it is not necessary for the cancellation algorithm. A heap expression is then represented via `term` as a list (seq) of elements. We could have represented the original heap expression more faithfully as a tree, but since $\cdot$ is commutative and associative, lists suffice for our purposes.

We will require two kinds of environments, which we package into the type of contexts:

$$\text{ctx} := \text{seq heap} \times \text{seq ptr}$$

The first component of a context is a list of heaps. In a term reflecting a heap expression, the element `Var n` stands for the $n$-th element of this list (starting from 0-th). Similarly, the second component is a list of pointers, and in the element `Pts m v`, the number $m$ stands for the $m$-th pointer in the list.

Because we will need to verify that our syntactic manipulation preserves the semantics of heap operations, we need a function that interprets syntax back into semantics. Assuming lookup functions `hlookup` and `plookup` which search for an index in a context of a heap or pointer, respectively, the interpretation function crawls over the syntactic term, replacing each number index with its value from the context (and returning an undefined heap, if the index is out of bounds). The function is implemented as follows:

$$\text{interp} (i : \text{ctx}) \ (t : \text{term}) : \text{heap} :=$$

$$\text{match} \ t \ \text{with}$$

$$\text{Var} \ n \ :: \ t' \ \Rightarrow \ \text{if} \ hlookup \ i \ n \ \text{is} \ \text{Some} \ h \ \text{then} \ h \bullet \text{interp} \ i \ t'$$

$$\text{else} \ \text{Undef}$$

$$| \ \text{Pts} \ m \ v \ :: \ t' \ \Rightarrow$$

$$\text{if} \ plookup \ i \ m \ \text{is} \ \text{Some} \ x \ \text{then} \ x \mapsto v \bullet \text{interp} \ i \ t'$$

$$\text{else} \ \text{Undef}$$

$$| \ \text{nil} \ \Rightarrow \ \text{empty}$$

end

For example, if the context $i$ is $([h_3, h_4], [x])$, then

$$\text{interp} \ i \ [\text{Pts} \ 0 \ v_1, \text{Var} \ 0, \text{Var} \ 1] = x \mapsto v_1 \bullet (h_3 \bullet (h_4 \bullet \text{empty}))$$

$$\text{interp} \ i \ [\text{Var} \ 1, \text{Pts} \ 0 \ v_2] = h_4 \bullet (x \mapsto v_2 \bullet \text{empty})$$
cancel \( (i : \text{ctx}) \ (t_1 \ t_2 \ r : \text{term}) : \text{Prop} := \)

match \( t_1 \) with
   nil \Rightarrow \text{interp} \ i \ r = \text{interp} \ i \ t_2 \)
| Pts \ m \ v :: \ t'_1 \Rightarrow \)
   if premove \( m \ t_2 \) \( \text{is Some} \) \( (v', t'_2) \) then cancel \( i \ t'_1 \ t'_2 \ r \land v = v' \)
   else cancel \( i \ t'_1 \ t_2 \) \( (\text{Pts} \ m \ v :: r) \)
| Var n :: \ t'_1 \Rightarrow \)
   if hremove \( n \ t_2 \) \( \text{is Some} \ t'_2 \) then cancel \( i \ t'_1 \ t'_2 \ r \)
   else cancel \( i \ t'_1 \ t_2 \) \( (\text{Var} \ n :: r) \)
end

Fig. 1. Heap cancellation algorithm.

Given this definition of \( \text{term} \), we can now encode the cancellation algorithm as a predicate (\textit{i.e.}, a function into \text{Prop}) in Coq (Figure 1). The predicate essentially constructs a conjunction of the residual equations obtained as a consequence of cancellation. Referring to Figure 1, the algorithm works as follows. It looks at the head element of the left term \( t_1 \), and tries to find it in the right term \( t_2 \) (keying on the index of the element). If the element is found, it is removed from both sides, before recursing over the rest of \( t_1 \). When removing a \text{Pts} element keyed on a pointer \( x \), the values \( v \) and \( v' \) stored into \( x \) in \( t_1 \) and \( t_2 \) must be equal. Thus, the proposition computed by \text{cancel} should contain an equation between these values as a conjunct. If the element is not found in \( t_2 \), it is shuffled to the accumulator \( r \), before recursing. When the term \( t_1 \) is exhausted, \textit{i.e.}, it becomes the empty list, then the accumulator stores the elements from \( t_1 \) that were not cancelled by anything in \( t_2 \). The equation between the interpretations of \( r \) and \( t_2 \) is a semantic consequence of the original equation, so \text{cancel} immediately returns it (our actual implementation performs some additional optimization before returning). The helper function \text{premove} \( m \ t_2 \) searches for the occurrences of the pointer index \( m \) in the term \( t_2 \) and, if found, returns the value stored into the pointer, as well as the term \( t'_2 \) obtained after removing \( m \) from \( t_2 \). Similarly, \text{hremove} \( n \ t_2 \) searches for \text{Var} \( n \) in \( t_2 \) and, if found, returns \( t'_2 \) obtained from \( t_2 \) after removal of \( n \).

Soundness of \text{cancel} is established by the following lemma which shows that the facts computed by \text{cancel} do indeed follow from the input equation between heaps, when \text{cancel} is started with the empty accumulator.

\[
\text{cancel\_sound} : \forall i : \text{ctx}. \forall t_1 \ t_2 : \text{term}.
\text{def} (\text{interp} \ i \ t_1) \rightarrow \text{interp} \ i \ t_1 = \text{interp} \ i \ t_2 \rightarrow \text{cancel} \ i \ t_1 \ t_2 \ \text{nil}
\]

The proof of \text{cancel\_sound} is rather involved and interesting in its own right, but we omit it here not to distract the attention of the reader from our main topic. (The interested reader can find it in our source files.) We could have proved the converse direction as well, to obtain a completeness result, but this was not necessary for our purposes.

The related work on proofs by reflection usually implements the cancellation phase in a manner similar to that above (see for example the work of Grégoire and Mahboubi, 2005). Where we differ from the related work is in the implementation of the reflection phase. This
phase is usually implemented by a tactic, but here we show that it can be implemented with canonical structures instead.

4.2 Reflection via canonical structures

Intuitively, the reflection algorithm traverses a heap expression, and produces the corresponding syntactic term. In our overloaded lemma, presented further below, we will invoke this algorithm twice, to reflect both sides of the equation. To facilitate cancellation, we need to ensure that identical variables on the two sides of the equation, call them $E_1$ and $E_2$, are represented by identical syntactic elements. Therefore, reflection of $E_1$ has to produce a context of newly encountered elements and their syntactic equivalents, which is then fed as an input to the reflection of $E_2$. If reflection of $E_2$ encounters an expression which is already in the context, the expression is reflected with the syntactic element provided by the context.

Notational Convention 3

Hereafter, projections out of an instance are considered implicit coercions, and we will typically omit them from our syntax. For example, in Figure 2 (described below), the canonical instance union_struct says union_tag ($f_1 \bullet f_2$) instead of union_tag ((untag (heap_of $f_1$)) $\bullet$ (untag (heap_of $f_2$))), which is significantly more verbose. This is a standard technique in Coq.

The reflection algorithm is encoded using the structure ast from Figure 2. The inputs to each traversal are the initial context $i$ of ast, and the initial heap in the heap_of projection. The output is the (potentially extended) context $j$ and the syntactic term $t$ that reflects the initial heap. One invariant of the structure is precisely that the term $t$, when interpreted under the output heap $j$, reflects the input heap:

$$\text{interp } j t = \text{heap_of}$$

There are two additional invariants needed to carry out the proofs:

$$\text{subctx } i j \quad \text{and} \quad \text{valid } j t$$

The first one states that the output context $j$ is an extension of the input context $i$, while the second one ensures that the syntactic term $t$ has no indices out of the bounds of the output context $j$. (We omit the definition of subctx and valid, but they can be found in the source files.)

There are several cases to consider during a traversal, as shown by the canonical instances in Figure 2. We first check if the input heap is a union, as can be seen from the ordering of tag synonyms (which is reversed, as demanded by the tagging pattern). In this case, the canonical instance is union_struct. The instance specifies that we recurse over both subheaps, by unifying the left subheap with $f_1$ and the right subheap with $f_2$.

The types of $f_1$ and $f_2$ show that the two recursive calls work as follows. First the call to $f_1$ starts with the input context $i$ and computes the output context $j$ and term $t_1$. Then the call to $f_2$ proceeds with input context $j$, and computes outputs $k$ and $t_2$. The output context of the whole union is $k$, and the output reflected term is the list-concatenation of $t_1$ and $t_2$. 

https://doi.org/10.1017/S0956796813000051 Published online by Cambridge University Press
The inputs to the structure are the parameter of an element, relying on unification to make syntactic comparisons between expressions.

When reflecting the empty heap, the instance is empty_struct. In this case, the input context \( i \) is simply returned as output, and the reflected term is the empty list.

When reflecting a singleton heap \( x \mapsto v \), the corresponding instance is ptr_struct. In this case, we first have to check if \( x \) is a pointer that already appears in the pointer part \( xs_1 \) of the input context. If so, we should obtain the index \( m \) at which \( x \) appears in \( xs_1 \). This is the number representing \( x \), and the returned reflected elem is \( \text{Pts} \ m \ v \). On the other hand, if \( x \) does not appear in \( xs_1 \), we need to add it. We compute a new context \( xs_2 \) which appends \( x \) at the end of \( xs_1 \), and this is the output pointer context for ptr_struct. The number \( m \) representing \( x \) in \( xs_2 \) now equals the size of \( xs_1 \), and returned reflected elem is again \( \text{Pts} \ m \ v \). Similar considerations apply in the case where we are reflecting a heap variable \( h \). The instance is then var_struct and we search in the heap portion of the context \( hs_1 \), producing a new heap portion \( hs_2 \).

In both cases, the task of searching (and possibly extending) the context is performed by the polymorphic structure xfind (Figure 3), which recurses over the context lists in search of an element, relying on unification to make syntactic comparisons between expressions. The inputs to the structure are the parameter \( s \), which is the sequence to search in, and the field elem_of, which is the (tagged) element to search for. The output sequence \( r \) equals \( s \) if elem_of is in \( s \), or extends \( s \) with elem_of otherwise. The output parameter \( i \) is the position at which the elem_of is found in \( r \).

If the searched element \( x \) appears at the head of the list, the selected instance is found_struct and the index \( i = 0 \). Otherwise, we recurse using recurse_struct. Ultimately, if \( s \) is empty, the returned \( r \) is the singleton \([x]\), via the instance extend_struct.
structure xtagged $A := \text{XTag} \{\text{xuntag} : A\}$

$\text{extend}\_\text{tag} \ A (x : A) := \text{XTag} \ x$
$\text{recurse}\_\text{tag} \ A (x : A) := \text{extend}\_\text{tag} \ x$
$\text{canonical}\ found\_\text{tag} \ A (x : A) := \text{recurse}\_\text{tag} \ x$

structure xfind $A (s r : \text{seq} \ A) (i : \text{nat}) :=$
$\text{XFind} \ \{\ \text{elem}_\text{of} : \text{xtagged} \ A;\$
$\_ : \text{index}_\text{r} \ i = \text{elem}_\text{of}_\wedge \text{prefix} \ s \ r \} \$

$\text{canonical}\ \text{found}\_\text{struct} \ A (x : A) (s : \text{seq} \ A) :=$
$\text{XFind} \ (x :: s) (x :: s) 0 (\text{found}\_\text{tag} \ x) \ldots$

$\text{canonical}\ \text{recurse}\_\text{struct} \ (i : \text{nat}) (y : A) (s r : \text{seq} \ A) \$
$\ (f : \text{xfind} \ s r i) :=$
$\text{XFind} \ (y :: s) (y :: r) (i + 1) (\text{recurse}\_\text{tag} \ f) \ldots$

$\text{canonical}\ \text{extend}\_\text{struct} \ A (x : A) :=$
$\text{XFind} \ \text{nil} \ [x] 0 (\text{extend}\_\text{tag} \ x) \ldots$

Fig. 3. Structure xfind for searching for an element in a list, and appending the element at the end of the list if not found.

It may be interesting to notice here that while xfind is in principle similar to find from Section 2.3, it is keyed on the element being searched for, rather than on the list (or in the case of find, the heap) in which the search is being performed. This exemplifies that there are many ways in which canonical structures of similar functionality can be organized. In particular, which term one keys on (i.e., which term one unifies with the projection from the structure) may in general depend on when a certain computation needs to be triggered. If we reorganized xfind to match find in this respect, then the structure ast would have to be reorganized too. Specifically, ast would have to recursively invoke xfind by unifying it against the contexts $x_1$ and $h_1$ in the instances ptr_struct and var_struct, respectively. As we will argue in Section 6, such unification can lead to incorrect results, if done directly, but we will be able to perform it indirectly, using a new design pattern.

We are now ready to present the overloaded lemma cancelR.

cancelR : $\forall \ j \ k : \text{ctx}. \forall _1 t_2 : \text{term}.$
$\forall f_1 : \text{ast} \ \text{nil} \ j \ t_1. \forall f_2 : \text{ast} \ j \ k \ t_2.$
$\text{def} \ (\text{untag} \ (\text{heap}\_\text{of} \ f_1)) \rightarrow$
$\text{untag} \ (\text{heap}\_\text{of} \ f_1) = \text{untag} \ (\text{heap}\_\text{of} \ f_2) \rightarrow$
$\text{cancel} \ k \ t_1 \ t_2 \ \text{nil}$

At first sight it may look strange that we are not using the notation convention 3 presented in this very same section. It is true that we can omit the projections and write $\text{def} \ f_1$ as the first hypothesis, but for the second one we have to be verbose. The reason is simple: if we write instead $f_1 = f_2$, Coq will consider this an equality on asts and not expand the implicit coercions, as needed.
Assuming we have a hypothesis

\[
H : x \mapsto v_1 \cdot (h_3 \cdot h_4) = h_4 \cdot x \mapsto v_2
\]

and a hypothesis \(D : \text{def } h_1\), we can *forwardly* apply cancelR to \(H\) using \(D\), *i.e.*,

\[
\text{move/(cancelR } D) : H
\]

This will make Coq fire the following unification problems:

1. \(\text{def } (\text{untag } (\text{heap_of } ?f_1)) \equiv \text{def } h_1\)
2. \(\text{untag } (\text{heap_of } ?f_1) \equiv h_1\)
3. \(\text{untag } (\text{heap_of } ?f_2) \equiv h_2\)

Because \(?f_1\) and \(?f_2\) are variable instances of the structure \(\text{ast}\), Coq will construct canonical values for them, thus reflecting the heaps into terms \(t_1\) and \(t_2\), respectively. The reflection of \(h_1\) will start with the empty context, while the reflection of \(h_2\) will start with the output context of \(f_1\), which in this case is \(([h_3, h_4], [x])\).

Finally, the lemma will perform cancel on \(t_1\) and \(t_2\) to produce \(v_1 = v_2 \land h_3 = \text{empty}\land\text{empty} = \text{empty}\). The trailing \text{empty} = \text{empty} can ultimately be removed with a few simple optimizations of cancel, which we have omitted to simplify the presentation.

**To reflect or not to reflect.** We used the power of canonical structures to inspect the structure of terms, in this case heaps, and then create abstract syntax trees representing those terms. Thus, a natural question arises: can’t we implement the cancellation algorithm entirely with canonical structures, just as it was implemented originally in one big tactic? The answer is yes, and the interested reader can find the code in the source files. There, we have encoded a similar algorithm to the one shown in Figure 1 with a structure parameterized over its inputs: the heaps on the left and on the right of the equation, the heap with “the rest”—*i.e.*, what could not be cancelled—and two invariants stating that (a) the right heap is defined and (b) the union of the left and rest heaps is equal to the right heap. As output, it returns a proposition and a proof of this proposition. Each instance of this structure will correspond to one step of the algorithm. As an immediate consequence, this direct encoding of the cancellation algorithm avoids the long proof of soundness, since each step (or instance) includes the local proof for that step only, and without having to go through the fuss of the interpretation of abstract syntax.

However, this approach has at least one big disadvantage: by comparison to the automated proof by reflection, it is slow. Indeed, this should come as no surprise, as it is well known that the method of proof by reflection is fast (*e.g.*, Grégoire & Mahboubi, 2005), and this is why we pursued it in the first place.

### 4.3 Dealing with heterogeneous heaps

So far we have represented singleton heaps as \(x \mapsto v\), assuming that all values in the heaps are of the same type. However, as promised above, we would like to relax this assumption. Allowing for variation in the type \(T\) of \(v\), a more faithful representation would be \(x \mapsto_T v\).

One easy way of supporting this without modifying our cancellation algorithm at all is to
view the values in the heap as being elements of type dynamic, \( i.e., \) as dependent pairs (or structures) packaging together a type and an element of the type:

\[
\text{structure dynamic := dyn \{ typ : Type; val : typ \}}
\]

In other words, we can simply model \( x \mapsto T v \) as \( x \mapsto \text{dyn } T v \). As a result, when we apply the cancelR lemma to the singleton equality \( x \mapsto T_1 v_1 = x \mapsto T_2 v_2 \), we would obtain:

\[
\text{dyn } T_1 v_1 = \text{dyn } T_2 v_2 \tag{3}
\]

But if the types \( T_1 \) and \( T_2 \) are equal, we would like to also automatically obtain the equality on the underlying values:

\[
v_1 = v_2 \tag{4}
\]

(Note that this cannot be done within the pure logic of Coq since equality on types is not decidable.) A key benefit of obtaining the direct equality on \( v_1 \) and \( v_2 \), rather than on \( \text{dyn } T_1 v_1 \) and \( \text{dyn } T_2 v_2 \), is that such an equality can then be fed into the standard rewrite tactic in order to rewrite occurrences of \( v_1 \) in a goal to \( v_2 \).

This effect could perhaps be achieved by a major rewriting of cancelR, but that would require a major effort, just to come up with a solution that is not at all modular. Instead, we will show now how we can use a simple overloaded lemma to post-process the output of the cancellation algorithm, reducing equalities between dynamic packages to equalities on their underlying value components wherever possible.

Figure 4 presents the algorithm to simplify propositions using canonical instances. Basically, it inspects a proposition, traversing through a series of \( \wedge \)s until it finds an equality of the form \( \text{dyn } A v_1 = \text{dyn } A v_2 \)—\( i.e., \) where the values on both sides have the same type \( A \)—in which case it returns \( v_1 = v_2 \). For any other case, it just returns the same proposition. We only consider the connective \( \wedge \) since the output of cancelR contains only this connective. If necessary, we could easily extend the algorithm to consider other connectives.
The algorithm is encoded in a structure called simplifier with three canonical instances, one for each of the aforementioned cases. The proof component of this structure lets us prove the following overloaded lemma. (Following notational convention 3, we omit the projectors, so $g$ below should be read as “the proposition in $g$”.)

$$\text{simplify} : \forall p : \text{Prop}. \forall g : \text{simplifier} p. \\
g \rightarrow p$$

By making $p$ and $g$ implicit arguments, we can write $\text{simplify} P$ to obtain the simplified version of $P$. For example, say we have to prove some goal with hypotheses

- $D: \text{def } (x \mapsto_T v_1 \bullet h_1)$
- $H: x \mapsto_T v_1 \bullet h_1 = h_2 \bullet x \mapsto_T v_2$

We can forwardly apply the composition of $\text{simplify}$ and $\text{cancelR } D$ to $H$

$$\text{move} : (\text{simplify} (\text{cancelR } D H))$$

to get the hypothesis exactly as we wanted:

$$v_1 = v_2 \land h_1 = h_2$$

## 5 Solving for functional instances

Previous sections described examples that search for a pointer in a heap expression or for an element in a list. The pattern we show in this section requires a more complicated functionality, which we describe in the context of our higher-order implementation of separation logic (Reynolds, 2002) in Coq. Interestingly, this search-and-replace pattern can also be described as higher-order, as it crucially relies on the type checker’s ability to manipulate first-class functions and solve unification problems involving functions.

To set the stage, the formalization of separation logic that we use centers on the predicate

$$\text{verify} : \forall A. \text{prog } A \rightarrow \text{heap } \rightarrow (A \rightarrow \text{heap } \rightarrow \text{Prop}) \rightarrow \text{Prop}.$$ 

The exact definition of verify is not important for our purposes here, but suffice it to say that it encodes a form of Hoare-style triples. Given a program $e : \text{prog } A$ returning values of type $A$, an input heap $i : \text{heap}$, and a postcondition $q : A \rightarrow \text{heap } \rightarrow \text{Prop}$ over $A$-values and heaps, the predicate

$$\text{verify } e \ i \ q$$

holds if executing $e$ in heap $i$ is memory-safe, and either diverges or terminates with a value $y$ and heap $m$, such that $q \ y \ m$ holds.

Programs can perform the basic heap operations: reading and writing a heap location, allocation, and deallocation. In this section, we focus on the writing primitive; given $x : \text{ptr}$ and $v : A$, the program $\text{write } x \ v : \text{prog unit}$ stores $v$ into $x$ and terminates. We also require
the operation for sequential composition, which takes the form of monadic bind:

\[
\text{bind : } \forall A. B. \text{prog } A \rightarrow (A \rightarrow \text{prog } B) \rightarrow \text{prog } B
\]

We next consider the following provable lemma, which serves as a Floyd-style rule for symbolic evaluation of write.

\[
\text{bnd\_write : } \forall A. B. C. \forall x : \text{ptr} . \forall v : A. \forall w : C. \forall e : \text{unit} \rightarrow \text{prog } B.
\]

\[
\forall h : \text{heap}. \forall q : B \rightarrow \text{heap} \rightarrow \text{Prop}.
\]

\[
\text{verify } (e ()) (x \mapsto v \bullet h) q \rightarrow
\]

\[
\text{verify } (\text{bind } (\text{write } x v) e) (x \mapsto w \bullet h) q
\]

To verify \(\text{write } x v\) in a heap \(x \mapsto w \bullet h\), it suffices to change the contents of \(x\) to \(v\), and proceed to verify the continuation \(e\).

In practice, \text{bnd\_write} suffers from the same problem as \text{indom} and \text{noalias}, as each application requires the pointer \(x\) to be brought to the top of the heap. We would like to devise an automated version \text{bnd\_writeR}, but, unlike \text{indomR}, application of \text{bnd\_writeR} will not merely check if a pointer \(x\) is in the heap. It will remember the heap \(h\) from the goal and reproduce it in the premise, only with the contents of \(x\) in \(h\) changed from \(w\) to \(v\).

For example, applying \text{bnd\_writeR} to the goal

\[
G_1 : \text{verify } (\text{bind } (\text{write } x_2 4) e)
\]

\[
(i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2) \bullet (i_2 \bullet x_3 \mapsto 3))
\]

should return a subgoal which changes \(x_2\) in place, as in:

\[
G_2 : \text{verify } (e ()) (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4) \bullet (i_2 \bullet x_3 \mapsto 3)) q
\]

### 5.1 The “search-and-replace” pattern

Here is where functions come in. The \text{bnd\_writeR} lemma should attempt to infer a function \(f\) which represents a heap with a “hole”, so that filling the hole with \(x \mapsto w\) (i.e., computing \(f (x \mapsto w)\)) results in the heap from the goal. Then replacing \(w\) with \(v\) is computed as \(f (x \mapsto v)\).

For example, in \(G_1\) we want to “fill the hole” with \(x_2 \mapsto 2\), while in \(G_2\), we want to fill it with \(x_2 \mapsto 4\). Hence, in this case, the inferred function \(f\) should be:

\[
\text{fun } k. i_1 \bullet (x_1 \mapsto 1 \bullet k) \bullet (i_2 \bullet x_3 \mapsto 3)
\]

To infer \(f\) using canonical structures, we generalize it from a function mapping heaps to heaps to a function mapping a heap \(k\) to a \textit{structure}, partition \(k r\) (defined in Figure 5), with a heap projection \text{heap\_of} that is equal to \(k \bullet r\). This \text{heap\_of} projection will be used to trigger the search for the subheap that should be replaced with a hole. (The role of the additional heap parameter \(r\) will be explained later, but intuitively one can think of \(r\) as representing the rest of the heap, \textit{i.e.}, the “frame” surrounding the hole.)
structure tagged_heap := Tag {untag : heap}

right_tag (h : heap) := Tag h
left_tag h := right_tag h
canonical found_tag h := left_tag h

structure partition (k r : heap) :=
  Partition { _ : heap_of = k • r }

canonical found_struct k :=
  Partition k empty (found_tag k)...

canonical left_struct h r (f : ∀ k. partition k r) k :=
  Partition k (r • h) (left_tag (f k • h))...

canonical right_struct h r (f : ∀ k. partition k r) k :=
  Partition k (h • r) (right_tag (h • f k))...

Fig. 5. Structure partition for partitioning a heap into the part matching \( k \) and “the rest” \((r)\).

Because the range of \( f \) depends on the input \( k \), \( f \) must have a dependent function type, and the \( \text{bnd\_writeR} \) lemma looks as follows.

\[
\text{bnd\_writeR} : \forall A\  B\  C. \forall x : \text{ptr}. \forall v : A. \forall w : C. \forall e : \text{unit} \to \text{prog} B. \\
\forall q : B \to \text{heap} \to \text{Prop}.
\forall r : \text{heap}. \forall f : (\forall k : \text{heap}. \\text{partition k r}) . \\
\text{verify} (e ()) (f (x \mapsto v)) q \to \\
\text{verify} (\text{bind} (\text{write} x v) e) (f (x \mapsto w)) q
\]

As before, following notational convention 3, we have omitted the projections and written \( f (x \mapsto w) \) instead of \( \text{untag} (\text{heap\_of} (f (x \mapsto w))) \), and similarly in the case of \( x \mapsto v \).

When the \( \text{bnd\_writeR} \) lemma is applied to a goal of the form

\[
\text{verify} (\text{bind} (\text{write} x v) e) h q
\]

the type checker creates unification variables for each of the parameters of \( \text{bind\_writeR} \), and proceeds to unify the conclusion of the lemma with the goal, getting the equation

\[
\text{untag} (\text{heap\_of} (?f (x \mapsto ?w))) = h
\]

where \(?f\) has type \( \forall k : \text{heap. partition k ?r} \). (Note that, because Coq’s unification follows a strict left-to-right order, \( x \) is not a unification variable but the actual location being written to in the goal.)

This unification goal will prompt the search (using the instances in Figure 5) for a canonical solution for \(?f\) with the property that the heap component of \(?f (x \mapsto w)\) syntactically equals \( h \), matching exactly the order and the parenthesization of the summands in \( h \). We have three instance selectors: one for the case where we found the heap we are looking for, and two to recurse over each side of the \( \bullet \).
The reader may wonder why all the instances of partition take the $k$ parameter last, thus forcing the $f$ parameter in the latter two instances to be itself abstracted over $k$ as well. The reason is best illustrated by following a partial trace of Coq’s unification.

Suppose $h = h_0 \cdot x \mapsto z$. After trying and failing to unify $?f (x \mapsto ?w)$ with $h$ using the instances found_struct and left_struct, it will proceed to try to use the instance right_struct. More precisely, as described previously in Section 3.2, when solving the equation

$$\text{heap}_{\text{of}} (?f (x \mapsto ?w)) \equiv \text{right}_{\text{tag}} (h_0 \cdot x \mapsto z)$$

Coq will:

1. Build an instance

   $$\text{right}_{\text{struct}} ?h' ?r' ?f' ?k'$$

   with $?h', ?r', ?f', ?k'$ fresh unification variables, $?f'$ with type $\forall k. \text{partition} \ k \ ?r'$ and the rest with type heap.

2. Unify the type of the instance from (6) with the type of the expected instance (i.e., the argument of heap_of) in (5). We know that $?f$ has type $\forall k : \text{heap}, \text{partition} \ k \ ?r$, and therefore that $?f (x \mapsto ?w)$ has type $\text{partition} \ (x \mapsto ?w) \ ?r$. The type of (6) is $\text{partition} \ ?k' \ (?h' \cdot ?r')$. Putting it all together, we get the equation

   $$\text{partition} \ ?k' \ (?h' \cdot ?r') \equiv \text{partition} \ (x \mapsto ?w) \ ?r$$

3. Unify the heap_of projection of (6) with the right-hand side of (5), that is,

   $$\text{right}_{\text{tag}} (?h' \cdot ?f' ?k') \equiv \text{right}_{\text{tag}} (h_0 \cdot x \mapsto z)$$

4. Finally, unify the instances:

   $$?f (x \mapsto ?w) \equiv \text{right}_{\text{struct}} ?h' ?r' ?f' ?k'$$

Altogether, we get the following equations that Coq processes in order:

1. $?k' \equiv x \mapsto ?w$
2. $?h' \cdot ?r' \equiv ?r$
3. $?h' \equiv h_0$
4. $?f' ?k' \equiv x \mapsto z$
5. $?f (x \mapsto ?w) \equiv \text{right}_{\text{struct}} ?h' ?r' ?f' ?k'$

The first three equations are solved immediately. The fourth one, if we expand the implicit projection and instantiate the variables, is

$$\text{untag} \ (\text{heap}_{\text{of}} (?f' (x \mapsto ?w))) \equiv x \mapsto z$$

Solving this recursively (we give the details of this part of the trace at the end of the section), Coq instantiates the following variables:

1. $?r' = \text{empty}$
2. $?w = z$
3. $?f' = \text{found}_{\text{struct}}$
Note how the type of the instance found_struct actually matches the type of $?f'$, that is, $\forall k. \text{partition } k \text{ empty}$.

Coq now can proceed to solve the last equation, which after instantiating the variables is

$$?f (x \mapsto z) \cong \text{right_struct } h_0 \text{ empty } \text{found_struct } (x \mapsto z)$$

which means that it has to find a function for $?f$ such that, when given the singleton $x \mapsto z$, produces the instance on the right-hand side. As it is well known (Goldfarb, 1981), higher-order unification problems are in general undecidable, as they might have an infinite number of solutions, without any one being the most general one. For this example, Coq takes a commonly-used pragmatic solution of falling back to a kind of first-order unification: it tries to unify the functions and then the arguments on both sides of the equation, which in this case immediately succeeds:

1. $?f \cong \text{right_struct } h_0 \text{ empty } \text{found_struct}$
2. $(x \mapsto z) \cong (x \mapsto z)$

This is the key to understanding why the instances of partition all take the $k$ parameter last: we want the $k$ parameter to ultimately be unified with the argument of $?f$. If the $k$ parameter did not come last, then Coq would try here to unify $x \mapsto z$ with whatever parameter did come last, which would clearly lead to failure.

Thus far, we have described how to construct the canonical solution of $?f$, but the mere construction is not sufficient to carry out the proof of $\text{bnd_writeR}$. For the proof, we further require an explicit invariant that $?f (x \mapsto v)$ produces a heap in which the contents of $x$ is changed to $v$, but everything else is unchanged when compared to $?f (x \mapsto w)$.

This is the role of the parameter $r$, which is constrained by the invariant in the definition of partition to equal the “rest of the heap”, that is

$$h = k \bullet r$$

With this invariant in place, we can vary the parameter $k$ from $x \mapsto w$ in the conclusion of $\text{bnd_writeR}$ to $x \mapsto v$ in the premise. However, $r$ remains fixed by the type of $?f$, providing the guarantee that the only change to the heap was in the contents of $x$.

It may be interesting to note that, while our code computes an $?f$ that syntactically matches the parentheses and the order of summands in $h$ (as this is important for using the lemma in practice), the above invariant on $h$, $k$ and $r$ is in fact a semantic, not a syntactic, equality. In particular, it does not guarantee that $h$ and $k \bullet r$ are constructed from the same exact applications of $\mapsto$ and $\bullet$, since in HTT those are defined functions, not primitive constructors. Rather, it captures only equality up to commutativity, associativity and other semantic properties of heaps as partial maps. This suffices to prove $\text{bnd_writeR}$, but more to the point: the syntactic property, while true, cannot even be expressed in Coq’s logic, precisely because it concerns the syntax and not the semantics of heap expressions.

To conclude the section, we note that the premise and conclusion of $\text{bnd_writeR}$ both contain projections out of $?f$. As a result, the lemma may be used both in forward reasoning

---

3 There exists a decidable fragment of higher-order unification called the “pattern fragment” (Miller, 1991). If the problem at hand falls into this fragment, Coq will find the most general unifier. However, our example does not fall into this fragment.
G. Gonthier et al.

(out of hypotheses) and in backward reasoning (for discharging a given goal). For example, we can prove the goal

\[
\text{verify } (\text{bind} \ (\text{write} \ x_2 \ 4) \ e) \ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2)) \ q
\]

under hypothesis

\[
H : \text{verify} \ (e ()) \ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4)) \ q
\]

in two ways:

- **Backward**: By applying \texttt{bnd\_writeR} to the goal. The goal will therefore be changed to exactly match \(H\).
- **Forward**: By applying \texttt{bnd\_writeR} \((x := x_2) \ (w := 2)\) to the hypothesis \(H\), thus obtaining the goal. Note how in this case we need to explicitly provide the instantiations of the parameters \(x\) and \(w\) because they cannot be inferred just from looking at \(H\).

This kind of versatility is yet another advantage that lemmas based on canonical instances exhibit when compared to tactics. The latter, it seems, must be specialized to either forward or backward mode, and we have not managed to encode a tactic equivalent of \texttt{bnd\_writeR} that is usable in both directions. It likewise appears difficult, if not impossible, to encode this bidirectional functionality using the style of proof by reflection we explored in Section 4.

**Fleshing out the trace.** In the partial unification trace given above, we streamlined the presentation by omitting the part concerning the resolution of the unification equation (7). Since \texttt{found\_tag} is the canonical instance of the \texttt{tagged\_heap} structure, instance resolution will reduce this problem to:

\[
\text{heap\_of} \ (?f' (x \mapsto ?w)) \equiv \text{found\_tag} \ (x \mapsto z) \quad (8)
\]

For this equation, Coq follows the same steps as in the processing of equation (5). It will:

1. Create a unification variable \(?k''\) for the argument of \texttt{found\_struct}.
2. Unify the type of \texttt{found\_struct} \(?k''\) with the type of \(?f' (x \mapsto ?w)\), that is,

\[
\text{partition} \ ?k'' \ \text{empty} \equiv \text{partition} (x \mapsto ?w) \ ?r'
\]

getting \(?k'' = x \mapsto ?w\) and \(?r' = \text{empty}\).
3. Unify the \texttt{heap\_of} projection of \texttt{found\_struct} \(?k''\) with the right-hand side of (8):

\[
\text{found\_tag} \ ?k'' \equiv \text{found\_tag} \ (x \mapsto z)
\]

Unfolding the already known definition of \(?k''\) as \(x \mapsto ?w\), we get \(?w = z\).
4. Unify the argument of \texttt{heap\_of} with the instance. After applying the solutions found so far, this produces

\[
?f' (x \mapsto z) \equiv \text{found\_struct} \ (x \mapsto z)
\]

As before, Coq solves this higher-order problem by unifying

(a) \(?f' \equiv \text{found\_struct}\)
(b) \(x \mapsto z \equiv x \mapsto z\)
5.2 Automatic lemma selection

In the previous section, we saw how to automate the symbolic evaluation of the command write. In our implementation of separation logic in Coq, there are in fact several such commands (allocation, deallocation, read, write, etc.), each of which may appear in one of two contexts (either by itself or as part of a bind expression sequencing it together with other commands). We have created one automated lemma for each combination of command and context, but picking the right one to apply at each step of a proof can be tedious, so we would like to automate this process by creating a procedure for selecting the right lemma in each case. We will now show how to build such an automated procedure.

For the running example we will use just two lemmas:

\[ \text{bnd}\text{\_writeR} : \forall A B C. \forall x : \text{ptr}. \forall v : A. \forall w : C. \forall e : \text{unit} \to \text{prog} B. \]
\[ \forall q : B \to \text{heap} \to \text{Prop}. \]
\[ \forall r : \text{heap}. \forall f : (\forall k : \text{heap}. \text{partition} k r). \]
\[ \text{verify} (e ()) (f (x \mapsto v)) q \to \]
\[ \text{verify} (\text{bind} (\text{write} x v) e) (f (x \mapsto w)) q \]

\[ \text{val}\text{\_readR} : \forall A. \forall x : \text{ptr}. \forall v : A. \forall q : A \to \text{heap} \to \text{Prop}. \]
\[ \forall r : \text{heap}. \forall f : (\text{partition} (x \mapsto v) r). \]
\[ (\text{def} f \to q v f) \to \]
\[ \text{verify} (\text{read} x) f q \]

The first one is the automated lemma from the previous section. The second one executes the command read alone (not as part of a bind expression). The lemma \text{val}\text{\_readR} states that, in order to show that read \(x\) satisfies postcondition \(q\), we need to prove that \(q\) holds of the value \(v\) that \(x\) points to.

Consider a simple example, in which we have the following goal:

\[ \text{verify} (\text{bind} (\text{write} x 4) (\text{fun} \_ \to \text{read} x)) (x \mapsto 0) (\text{fun} r \_ \to r = 4) \]

This goal states that after writing the value 4 in location \(x\), we read \(x\) and get a result \(r\) that is equal to 4. Using the above lemmas directly, we would prove this as follows:\footnote{Note that we are using \text{Ssreflect apply: (i.e., with colon)} instead of Coq’s native tactic. This is required since Coq’s apply tactic might use two different and inconsistent unification algorithms.}

\[ \text{apply: bnd}\text{\_writeR} \]
\[ \text{by apply: val}\text{\_readR} \]

The first application changes the goal to

\[ \text{verify} (\text{read} x) (x \mapsto 4) (\text{fun} r \_ \to r = 4) \]

while the second application performs the read and produces the trivial goal \(4 = 4\).

Using the overloaded lemma \text{step} we will present below, we will instead be able to prove this as follows:

\[ \text{by do 2 apply: step} \]
structure val_form \( h : \text{heap} \) \((q : A \rightarrow \text{heap} \rightarrow \text{Prop}) (p : \text{Prop}) \):=
ValForm \{ val\_pivot : \text{prog} \ A ;
\_ : p \rightarrow \text{verify val\_pivot} \ h \ q \}

structure bnd_form \( h : \text{heap} \) \((e : A \rightarrow \text{prog} \ B) (q : B \rightarrow \text{heap} \rightarrow \text{Prop}) (p : \text{Prop}) \):=
BndForm \{ bnd\_pivot : \text{prog} \ A ;
\_ : p \rightarrow \text{verify (bind bnd\_pivot} \ e \ h \ q \}

canonical val\_bind\_struct \( h \ e \ q \ p \ (f : \text{bnd\_form} \ h \ e \ q \ p) \):=
ValForm (bind \( \text{bnd\_pivot} \ f \) \ e) ...

\begin{align*}
\text{step} & : \forall h. \forall q. \forall p. \forall f : \text{val\_form} \ h \ q \ p. \ p \rightarrow \text{verify} \ f \ h \ q
\end{align*}

Fig. 6. Definition of the overloaded step lemma.

where do \( n \) \( \text{tactic} \) is the \( n \)th repeated application of \( \text{tactic} \). When verifying a large program, an overloaded lemma like \( \text{step} \) becomes increasingly convenient to use, for all the usual reasons that overloading is useful in ordinary programming. This convenience is borne out in our source files, where the interested reader can find the verification of a linked list data type using \( \text{step} \).

Intuitively, \( \text{step} \) works by inspecting the program expression being verified and selecting an appropriate lemma to apply. In our example, the first application of \( \text{step} \) will apply \( \text{bnd\_writeR} \), and the second one \( \text{val\_readR} \), exactly as in our manual proof.

Notational Convention 4

We use \( \text{val\_*} \) to name every lemma concerning the symbolic execution of a program expression consisting of a single command, like \( \text{val\_readR} \). We use \( \text{bnd\_*} \) to name every lemma concerning the symbolic execution of a command inside a bind expression, like \( \text{bnd\_writeR} \).

Figure 6 shows the main structure, \( \text{val\_form} \), for the overloaded lemma \( \text{step} \). It has two fields: (1) a program expression, which we call \( \text{val\_pivot} \), and (2) a proof that, assuming precondition \( p \), the postcondition \( q \) will hold after executing the pivot program in the initial heap \( h \). Our overloaded lemma \( \text{step} \) is trivially proved by the projection of this second field.

When \( \text{step} \) is applied to a goal \( \text{verify} \ e \ h \ q \), the system tries to construct an instance \( f : \text{val\_form} \ h \ q \ p \), whose \( \text{val\_pivot} \) matches \( e \). Figure 7 declares one such \( \text{val\_*} \) instance, \( \text{val\_read\_struct} \), which is selected when \( e \) is a read \( x \) command, for some \( x \). The second field of the instance declares the lemma that should be applied to the \( \text{verify} \) goal; in this case the \( \text{val\_readR} \) lemma.

Thus, declaring instances of \( \text{val\_form} \) corresponds to \( \text{registering} \) with \( \text{step} \) the lemmas that we want applied for specific forms of \( e \). For example, we can register lemmas that apply when \( e \) is a primitive command such as alloc or dealloc. The only requirement is that the form of the registered lemma matches the second field of \( \text{val\_form} \); namely, the lemma has a conclusion \( \text{verify} \ e \ h \ q \) and \( \text{one} \) premise \( p \), for some \( e, h, q, \) and \( p \).

When the command \( e \) is not a stand-alone command, but a bind composite, we redirect \( \text{step} \) to search for an appropriate lemma among \( \text{bnd\_*} \) instances. That is achieved by declaring a new structure \( \text{bnd\_form} \) and a canonical instance \( \text{val\_bind\_struct} \) for \( \text{val\_form} \) in Figure 6. When \( \text{step} \) is applied to a goal verify \( e \ h \ q \), in which \( e \) is a bind, \( \text{val\_bind\_struct} \) is
How to make ad hoc proof automation less ad hoc

387

canonical val_read_struct x v q r f :=
ValForm (read x) (val_readR x v q r f)

canonical bnd_write_struct x v w e q r f :=
BndForm (write x v) (bnd_writeR x v w e q r f)

Fig. 7. Registering individual lemmas with step.

selected as a canonical instance. But since val_bnd_struct is parameterized by a hypothesis $f : bnd\_form \ h e q p$, this redirects the unification algorithm into solving for $f$.

Much as in the val_form case, we need to register the bnd_* lemmas that the algorithm should apply depending on the first command of $e$. Figure 7 shows an example instance bnd_write_struct which registers lemma bnd_writeR to be applied by step whenever $e$ starts with a write command.

In a similar way, we can register val_* and bnd_* lemmas for user-defined commands as well, thereby extending step at will as we implement new commands. In this sense, step is an open-ended automation procedure. Such open-endedness is yet another aspect of lemma overloading that does not seem to have a direct correspondent in tactic-based automation.

6 Flexible composition and application of overloaded lemmas

In this section, we construct an overloaded version of the noalias lemma from Section 1. This example presents two main challenges: (1) composing two overloaded lemmas where the output of one is the input to the other one, and (2) making the resulting lemma applicable in both forward and backward reasoning.

Concerning the first problem, there are several ways to solve it. We present different alternative approaches to composing overloaded lemmas, equipping the interested reader with a handy set of techniques with varying complexity/flexibility tradeoffs.

Concerning the second problem, the key challenge is to ensure that the unification constraints generated during canonical structure inference are resolved in the intended order. This is important because the postponing of a certain constraint may underspecify certain variables, leading the system to choose a wrong intermediate value that will eventually fail to satisfy the postponed constraint. In the case of noalias, the problem is that a naive implementation will result in the triggering of a search for a pointer in a heap before we know what pointer we are searching for. Fortunately, it is possible to handle this problem very easily using a simple design pattern we call parameterized tagging.

In the following sections, we build several, progressively more sophisticated, versions of the noalias lemma, ultimately arriving at a lemma noaliasR that will be applicable both backwards and forwards. In the backward direction, we will be able to use to it solve a goal such as

\[(x_1 \neq x_2) \land (x_2 \neq x_3) \land (x_3 \neq x_1)\]

by rewriting repeatedly (notice the modifier “!”):

\[
\text{by rewrite !(noaliasR } D)\]

Here, $D$ is assumed to be a hypothesis describing the well-definedness of a heap containing three singleton pointers, one for each pointer appearing in the goal. Notice how, in order
Fig. 8. Structure scan for computing a list of pointers syntactically appearing in a heap.

to rewrite repeatedly using the same lemma (noaliasR D), it is crucial that we do not need to explicitly specify the pointers involved in each application of the lemma, since each application involves a different pair of pointers. In our first versions of the lemma, this advanced functionality will not be available, and the pointers will need to be given explicitly, but eventually we will show how to support it.

Before exploring the different versions of the lemma, we begin by presenting the infrastructure common to all of them.

6.1 Basic infrastructure for the overloaded lemma

Given a heap \( h \), and two pointers \( x \) and \( y \), the algorithm for noalias proceeds in three steps: (1) scan \( h \) to compute the list of pointers \( s \) appearing in it, which must by well-definedness of \( h \) be a list of distinct pointers; (2) search through \( s \) until we find either \( x \) or \( y \); (3) once we find one of the pointers, continue searching through the remainder of \( s \) for the other one. Figures 8–10 show the automation procedures for performing these three steps.

Step (1) is implemented by the scan structure in Figure 8. Like the ast structure from Section 4, scan returns its output using its parameter (here, \( s \)). It also outputs a proof that the pointers in \( s \) are all distinct (i.e., uniq \( s \)) and that they are all in the domain of the input heap, assuming it was well-defined.

Step (2) is implemented by the search2 structure (named so because it searches for two pointers, both taken as parameters to the structure). It produces a proof that \( x \) and \( y \) are both distinct members of the input list \( s \), which will be passed in through unification with the seq2_of projection. The search proceeds until either \( x \) or \( y \) is found, at which point the search1 structure (next paragraph) is invoked with the other pointer.

Step (3) is implemented by the search1 structure, which searches for a single pointer \( x \) in the remaining piece of \( s \), returning a proof of \( x \)'s membership in \( s \) if it succeeds. Its implementation is quite similar to that of the find structure from Section 2.3.
structure tagged_seq2 := Tag2 {untag2 : seq ptr}
foundz (s : seq ptr) := Tag2 s
foundy s := foundz s
canonical foundx s := foundy s

structure search2 (x y : ptr) :=
Search2 { seq2_of : tagged_seq2;
  : x ∈ seq2_of ∧ y ∈ seq2_of
  ∧ (uniq seq2_of → x != y) }
canonical x_struct x y (s1 : search1 y) :=
Search2 x y (foundx (x :: s1)) ...
canonical y_struct x y (s1 : search1 x) :=
Search2 x y (foundy (y :: s1)) ...
canonical z_struct x y z (s2 : search2 x y) :=
Search2 x y (foundz (z :: s2)) ...

Fig. 9. Structure search2 for finding two pointers in a list.

structure tagged_seq1 := Tag1 {untag1 : seq ptr}
recurse_tag (s : seq ptr) := Tag1 s
canonical found_tag s := recurse_tag s

structure search1 (x : ptr) := Search1 { seq1_of : tagged_seq1;
  : x ∈ seq1_of }
canonical found_struct (x : ptr) (s : seq ptr) :=
Search1 x (found_tag (x :: s)) ...
canonical recurse_struct (x y : ptr) (f : search1 x) :=
Search1 x (recurse_tag (y :: f)) ...

Fig. 10. Structure search1 for finding a pointer in a list.

6.2 Naive composition

The noalias lemma we wish to build is, at heart, a composition of two subroutines: one implemented by the structure scan and the other by the structure search2. Indeed, looking at the structures scan and search2, we notice that the output of the first one coincides with the input of the second one: scan computes the list of distinct pointers in a heap, while search2 proves that, if the list output by scan contains distinct pointers, then the two pointers given as parameters are distinct.

Our first, naive attempt at building noalias is to (1) define overloaded lemmas corresponding to scan and search2, and (2) compose them using ordinary (function) composition, in the same that way that we composed the two lemmas simplify and cancelR in Section 4.3. As we will see, this direct approach does not quite work—i.e., we will not get out a general lemma in the end—but it is instructive to see why.

First, we create the two overloaded lemmas, scan_it and search_them, whose proofs are merely the proof projections from the scan and search2 structures. As usual, we leave the
structure projections (here of $f$ and $g$) implicit:

\[
\text{scan}_i : \forall s : \text{seq} \: \text{ptr} \cdot \forall f : \text{scan} \: s. \\
\text{def } f \rightarrow \text{uniq } s
\]

\[
\text{search}_\text{them} : \forall x \: y : \text{ptr} \cdot \forall g : \text{search}_2 \: x \: y. \\
\text{uniq } g \rightarrow x \neq y
\]

We can apply their composition in the same way as in Section 4.3. For example:

Hyp. $D$ : def $(i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))$  
Goal : $x_1 \neq x_3$  
Proof : by apply : $(\text{search}_\text{them} \: x_1 \: x_3 \: (\text{scan}_i \: D))$

During the typechecking of the lemma to be applied (i.e., $\text{search}_\text{them} \: x_1 \: x_3 \: (\text{scan}_i \: D)$), Coq will first unify $D$'s type with that of $\text{scan}_i$'s premise (i.e., def $(\text{untag } (\text{heap}_\text{of} \: ?f))$), which forces the unification of the heap in the type of $D$ with the implicit projection $\text{untag} (\text{heap}_\text{of} \: ?f)$. This in turn triggers an inference problem in which the system solves for the canonical implementation of $?f$ by executing the scan algorithm, thus computing the pointer list $s$ (in this case, $[x_1, x_2, x_3]$). After obtaining uniq $s$ as the output of $\text{scan}_i$, the search for the pointers $x_1$ and $x_3$ is initiated by unifying uniq $s$ with the premise of $\text{search}_\text{them}$ (i.e., uniq $(\text{seq}_2 \: \text{of} \: (\text{untag}_2 \: ?g))$), which causes $s$ to be unified with $\text{seq}_2 \: \text{of} \: (\text{untag}_2 \: ?g)$, thus triggering the resolution of $?g$ by the $\text{search}_2$ algorithm.

We may be tempted, then, to define the lemma noalias as a direct composition of the two lemmas. Unfortunately this will not work, because although we can compose the lemmas dynamically (i.e., when applied to a particular goal), we cannot straightforwardly compose them statically (i.e., in the proof of the general noalias lemma with unknown parameters). To see why, let us try to build the lemma using the structures:

\[
\text{noalias}_R \_\text{fwd} \_\text{wrong} : \forall x \: y : \text{ptr} \cdot \forall s : \text{seq} \: \text{ptr} \cdot \forall f : \text{scan} \: s. \forall g : \text{search}_2 \: x \: y. \\
\text{def } f \rightarrow x \neq y
\]

The reason we cannot prove this lemma is that there is no connection between the output of the scan—that is, $s$—and the input sequence of $\text{search}_2$. To put it another way, what we have is a proof that the list of pointers $s$ appearing in the heap component of $f$ is unique, but what we need is a way to prove that the list component of $g$ is unique. We do not have any information telling us that these two lists should be equal, and in fact there is no reason for that to be true.

### 6.3 Connecting the lemmas with an equality hypothesis

To prove the general noalias lemma, we clearly need a way to connect the output of scan with the input of $\text{search}_2$. A simple way to achieve this is by adding an extra hypothesis representing the missing connection (boxed below), using which the proof of the lemma is straightforward. For clarity, we make the projection in this hypothesis explicit.

\[
\text{noalias}_R \_\text{fwd} : \forall x \: y : \text{ptr} \cdot \forall s : \text{seq} \: \text{ptr} \cdot \forall f : \text{scan} \: s. \forall g : \text{search}_2 \: x \: y. \\
\text{def } f \rightarrow s = (\text{untag}_2 (\text{seq}_2 \: \text{of} \: g)) \rightarrow x \neq y
\]
We show how it works by applying it to the same example as before. We assume $s, f, g$ implicit.

Hyp. $D$:
$$\text{def } (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))$$

Goal : $x_1 \neq x_3$

Proof : by apply:\n$$\text{(noaliasR\_fwd} x_1 x_3 D \text{ (erefl _))}$$

Here, erefl $x$ is the proof that $x = x$. The trace of the type checker is roughly as follows. It:

2. Unifies $?x$ and $?y$ with the pointers given as parameters, $x_1$ and $x_3$, respectively.
3. Unifies the hypothesis $D$ with the hypothesis of the lemma. More precisely, with $?f : \text{scan } ?s$, it will unify
$$\text{def } ?f \overset{\Delta}{=} \text{def } (i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3))$$

starting the scan’ing of the heap. When scan is finished, we get $?s \overset{\Delta}{=} [x_1, x_2, x_3]$.
4. Unifies the equality with the type of erefl $\_\text{, where } \_\text{ is an implicit argument. More precisely, Coq will create a unification variable } ?t\text{ for this implicit argument, and unify}$
$$ (?t = ?t) \overset{\Delta}{=} ([x_1, x_2, x_3] = \text{untag2 (seq2\_of } ?g))$$

where $?g : \text{search2 } x_1 x_3$. Next, it decomposes the equality, obtaining equations

$$ ?t \overset{\Delta}{=} [x_1, x_2, x_3]$$
$$ ?t \overset{\Delta}{=} \text{untag2 (seq2\_of } ?g)$$

which effectively results in the equation we need for triggering the search for the two pointers in $s$:

$$[x_1, x_2, x_3] \overset{\Delta}{=} \text{untag2 (seq2\_of } ?g)$$

The astute reader may have noticed that, in the definition of noaliasR\_fwd, we purposely arranged for the hypothesis def $f$ to appear before the equality of $s$ and $g$. If instead we had put it afterwards, the last two steps would have been swapped, resulting in a doomed search for the pointers in $s$ before the identity of $s$ as $[x_1, x_2, x_3]$ was known.

In fact, note that we can actually arrange for def $f$ to be hoisted out even further, outside the scope of the pointers $x$ and $y$. The benefit of doing so is that we can scan a heap just once, and then use the resulting lemma to prove non-aliasing properties between different pairs of pointers without rescanning the heap each time. In particular, let our lemma be

$$\text{noaliasR\_fwd} : \forall s : \text{seq ptr. } \forall f : \text{scan s. } \forall d : \text{def } f. \forall x y : \text{ptr. } \forall g : \text{search2 } x y. $$
$$s = \text{untag2 (seq2\_of } g) \rightarrow x \neq y$$

and hypothesis $D$ be as before. We can then make a local lemma abbreviation with the partially applied lemma

$$\text{have } F := \text{noaliasR\_fwd } D$$

Typechecking this local definition implicitly solves for $s$ and $f$ by scanning the heap defined by $D$, leaving only $x, y$, and $g$ to be resolved by subsequent instantiation. We can
then solve the following goal by rewriting several times using the partially-applied \( F \):

\[
\begin{align*}
\text{Goal:} & \quad x_1 \neq x_3 \land x_2 \neq x_3 \\
\text{Proof:} & \quad \text{by rewrite } (F \ x_1 \ x_3 \ (\text{erefl} \ _)) \ (F \ x_2 \ x_3 \ (\text{erefl} \ _))
\end{align*}
\]

### 6.4 Looking backward, not forward

Note that, when applying the `noaliasR_fwd` lemma from the previous section, we need to instantiate the pointer parameters \( x \) and \( y \) explicitly, or else the search will fail. More precisely, the search will proceed without knowing which pointers to search for, and Coq will end up unifying \( x \) and \( y \) with (as it happens) the first two pointers in the list \( s \) (in the above example, \( x_1 \) and \( x_2 \)). If they are not the pointers from the goal (as indeed in the example they are not), the application of the lemma will simply fail.

For many examples, like the `cancelR` example in Section 4, this is not a problem since the lemma is intended to be used only in forward mode. However, if we want `noalias` to be applicable also in backward mode—in particular, if we want to rewrite repeatedly with `noalias D` as shown at the beginning of this section—then we need to find a way of helping Coq infer the pointer arguments. The approach we present in this section will demonstrate yet another way of composing lemmas, which improves on the approach of the previous section in that it enables one (but not both) of the pointer arguments of `noalias` to be inferred. It thus serves as a useful bridge step to the final version of `noalias` in Section 6.5, which will be applicable both forwards and backwards.

The idea is to replace the equality hypothesis in the previous formulation of the lemma with a bespoke structure, `check`, which serves to unify the output of `scan` and the input of `search2`. To understand where we put the “trigger” for this structure, consider the `noaliasR_fwd` lemma from the previous section. There, the trigger was placed strategically after the hypothesis `def f` to fire the search after the list of pointers is computed. If we want to remove the equality hypothesis, then we do not have any other choice but to turn one of the pointers into the trigger.

We show first the reformulated lemma `noaliasR_fwd` to explain the intuition behind this change. As in our last version of the lemma, we move the hypothesis `def f` before the pointer arguments \( x \) and \( y \) to avoid redundant recomputation of the `scan` algorithm.

```plaintext
noaliasR_fwd : \forall s : \text{seq ptr}. \forall f : \text{scan s}. \forall d : \text{def f}. \forall x y : \text{ptr}. \forall g : \text{check x y s}.
\quad x \neq y \ \text{of } g
```

As before, when the lemma is applied to a hypothesis \( D \) of type `def h`, the heap \( h \) will be unified with the (implicit) projection `untag (heap_of f)`. This will execute the `scan` algorithm, producing as output the pointer list \( s \). However, when this lemma is subsequently applied in order to solve a goal of the form \( x' \neq y' \), the unification of that goal with the conclusion of the lemma will trigger the unification of \( y' \) with `y_of ?g` (the projection from `check`, which we have made explicit here for clarity), and that will in turn initiate the automated search for the pointers in the list \( s \). Note that we could use the unification with either \( x' \) or \( y' \) to trigger the search, but here we have chosen the latter.
We define the structure check and its canonical instance start as follows:

```plaintext
structure check (xy : ptr) (s : seq ptr) :=
  Check { y.of : ptr;
    _ : y = y.of;
    _ : uniq s → x != y.of }

canonical start xy (s_2 : search2 xy) :=
  Check xy (untag2 (seq2_of s_2)) y (erefl _) ...
```

The sole purpose of the canonical instance start for the check structure is to take the pointers \( x' \) and \( y' \) and the list \( s \), passed in as parameters, and repackage them appropriately in the form that the search2 structure expects. In particular, while check is keyed on the right pointer (here, \( y' \)), search2 is keyed on the list of pointers \( s \), so a kind of coercion between the two structures is necessary. Notice that this coercion is only possible if the structure’s pointer parameter \( y \) is constrained to be equal to its \( y \).of projection. Without this constraint, appearing as the second field (first proof field) of check, we obviously cannot conclude \( x != y \).of from \( x != y \).

Knowing that Coq unifies subterms in a left-to-right order, it should be clear that we can avoid mentioning the pointer \( x' \), since Coq will unify ?\( x \) with \( x' \) before unifying \( y \).of \( g \) with \( y' \) and triggering the search. Consequently, if we have the goal

\[
x_1 != x_3 \land x_2 != x_3
\]

and \( D \) has the same type as before, we can solve the goal by repeated rewriting as follows:

```plaintext
rewrite !(noaliasR_fwd D \_ x_3)
```

It is thus sensible to ask if we can avoid passing in the instantiation for the \( y \) parameter (here, \( x_3 \)) explicitly. Unfortunately, we cannot, and the reason will become apparent by following a trace of application.

Assume \( D \) as before. We solve the goal

\[
x_1 != x_3
\]

by

```plaintext
apply: (noaliasR_fwd D \_ x_3)
```

For the application to succeed, it must unify the goal with the inferred type for the lemma being applied. Let us write \(?s, ?f, ?d, ?x, ?y \) and \(?g \) for the unification variables that Coq creates for the arguments of noaliasR_fwd. As usual in type inference, the type of \( D \) must match the type of \(?d, i.e., def \( ?f \). As mentioned above, this produces the unification problem that triggers the scanning:

\[
untag (heap_.of \?f) ≜ i_1 • (x_1 ↦ v_1 • x_2 ↦ v_2) • (i_2 • x_3 ↦ v_3)
\]

In solving this problem, Coq instantiates variables \(?f \) and \(?s \) with \( I_f \) and \([x_1,x_2,x_3] \) respectively, where \( I_f \) stands for an instance that we omit here.

Now Coq continues the inference, instantiating \(?y \) with \( x_3 \), and getting the following type for the conclusion of the lemma:

\[
?x != y_.of \?g
\]
where \( g : \text{check } ?x \, x_3 \, [x_1, x_2, x_3] \). Coq proceeds to unify this type with the goal:

\[
?x \neq \text{y._of } ?g \equiv x_1 \neq x_3
\]

This generates two subproblems:

1. \(?x \equiv x_1\)
2. \(\text{y._of } ?g \equiv x_3\)

The first one is solved immediately, and the second one triggers the search for an instance. The only available instance is \(\text{start} \). After creating unification variables \(?x'\), \(?y'\) and \(?s'_2\), one for each argument of the instance, Coq unifies the type of the instance with the expected type, \(i.e.,\)

\[
\text{check } ?x' \, ?y' \, (\text{untag2 } (\text{seq2._of } ?s'_2)) \equiv \text{check } x_1 \, x_3 \, [x_1, x_2, x_3]
\]

therefore creating the following unification problems:

1. \(?x' \equiv x_1\)
2. \(?y' \equiv x_3\)
3. \(\text{untag2 } (\text{seq2._of } ?s'_2) \equiv [x_1, x_2, x_3]\)

After solving the first two problems right away, the third problem triggers the search for the pointers in the list. It is only now, after successfully solving these equations, that Coq unifies the \(\text{y._of} \) projection of the instance with the expected one from the original equation, getting the trivial equation \(x_3 \equiv x_3\). Finally, it can assign the instance to \(?g\), \(i.e.,\)

\[
?g \equiv \text{start } x_1 \, x_3 \, ?s'_2
\]

thus successfully proving the goal.

It should now hopefully be clear why we needed to pass \(x_3\) as an explicit argument. If we had left it implicit, the unification problem \#3 above would have triggered a search on an unknown \(?y'\), even though the information about the identity of \(?y'\) was made available in the subsequent step.

### 6.5 Reordering unification subproblems via parameterized tagging

Fortunately, there is a simple fix to our check structure to make our lemma infer \textit{both} pointers from the goal without any intervention from the user.

**The pattern.** In order to fix our check structure, we need a way to reorder the unification subproblems so that \(?y\) gets unified with \(x_3\) before the search algorithm gets triggered on the pointer list \(s\). The trick for doing this is to embed the parameter \(y\) in the \textit{type} of the projector \(\text{y._of} \), thus ensuring higher priority in the unification order. Specifically, we will give \(\text{y._of} \) of the type \texttt{equals.ptr y}, which (as the name suggests) will serve to constrain \(\text{y._of} \) to be equal to \(y\) and to ensure that this constraint is registered \textit{before} the search algorithm is triggered. (Technically, \texttt{equals.ptr y} is \textit{not} a singleton type, but canonical instance resolution will cause it to effectively behave like one.) We call this pattern \textit{parameterized tagging}.
To illustrate, we present the structure equals_ptr, its canonical instance equate, and the requisite changes to the check structure (and its instance) according to the pattern:

```plaintext
structure equals_ptr (z : ptr) := Pack {unpack : ptr}
canonical equate (z : ptr) := Pack z z
structure check (x y : ptr) (s : seq ptr) :=
    Check { y_of : equals_ptr y;
            z : uniq s → x path un= unpack y_of }
canonical start x y (s2 : search2 x y) :=
    Check x y (un= tag2 (seq2_of s2)) (equate y) ...
```

Here, the instance equate guarantees that the canonical value of type equals_ptr z is a package containing the pointer z itself.

We can now revise our statement of the overloaded noaliasR_fwd lemma ever so slightly to mention the new projector unpack (which could be made implicit). We also rename it, since this is the final presentation of the lemma:

```plaintext
noaliasR : ∀s : seq ptr. ∀f : scan s. ∀d : def f. ∀x : ptr. ∀g : check x y s.
    x /= unpack (y_of g)
```

As before, suppose that noaliasR has already been applied to a hypothesis D of type def h, so that the lemma’s parameter s has already been solved for. Then, when noaliasR is applied to a goal x1 /= x3, the unification engine will unify x1 with the argument ?x of noaliasR, and proceed to unify

```
unpack (y_of ?g) ≡ x3
```

in a context where ?g : check x1 ?y s. In order to fully understand what is going on, we detail the steps involved in the instance search. The only instance applicable is equate.

After opening the instance by creating the unification variable ?z, Coq unifies the type of the instance with the type of y_of ?g:

```plaintext
equals_ptr ?z ≡ equals_ptr ?y
```

and obtains the solution ?z = ?y. Then, the unpack projection from equate ?z (which is simply ?z) is unified with the value it is supposed to match, namely x3. This step is the key to understanding how we pick up x3 from the goal. Replacing ?z with its solution ?y, we thus get the equation

```
?y ≡ x3
```

Finally, Coq unifies the expected instance with the one computed:

```
y_of ?g ≡ equate x3
```

which triggers the search for the pointers in s as before.

**Applying the lemma.** The overloaded noaliasR lemma supports a number of modes of use: it can be applied, used as a rewrite rule, or composed with other lemmas. For example,
G. Gonthier et al.

assume that we have a hypothesis specifying a disjointness of a number of heaps in a union:

\[
D : \text{def} \left( i_1 \mapsto (x_1 \mapsto v_1 \cdot x_2 \mapsto v_2) \cdot (i_2 \mapsto x_3 \mapsto v_3) \right)
\]

Assume further that the arguments \(x, y, s, f\) and \(g\) of noaliasR are implicit, so that we can write simply \((\text{noaliasR } D)\) when we want to partially instantiate the lemma with the hypothesis \(D\). Then the following are some example goals, and proofs to discharge them, illustrating the flexibility of use. As can be seen, no tedious reordering of heap expressions by commutativity and associativity is needed.

1. The lemma can be used in backward reasoning. The type checker picks up \(x_1\) and \(x_2\) from the goal, and confirms they appear in \(D\).

   Goal: \(x_1 \neq x_2\)
   Proof: by apply: \((\text{noaliasR } D)\)

2. The lemma can be used in iterated rewriting. The lemma is partially instantiated with \(D\). It performs the initial scan of \(D\) once, but is then used three times to reduce each conjunct to \(true\). There is no need in the proof to specify the input pointers to be checked for aliasing. The type checker can pick them up from the goal, in the order in which they appear in the conjunction.

   Goal: \((x_1 \neq x_2) \&\& (x_2 \neq x_3) \&\& (x_3 \neq x_1)\)
   Proof: by rewrite \((\text{noaliasR } D)\)

3. The lemma can be composed with other lemmas, to form new rewrite rules. Again, there is no need to provide the input pointers in the proofs. For example, given the standard library lemma negbTE: \(\forall b : \text{bool}. \! b \land \text{true} = \! b\), we have:

   Goal: if \((x_2 = x_3) \&\& (x_1 \neq x_2)\) then false else true
   Proof: by rewrite \((\text{negbTE } (\text{noaliasR } D))\)

4. That said, we can provide the input pointers in several ways, if we wanted to, which would correspond to forward reasoning. We can use the term selection feature of rewrite to reduce only the specified conjunct in the goal.

   Goal: \((\!(x_1 \neq x_2) \&\& (x_2 \neq x_3)) = (x_1 \neq x_2)\)
   Proof: by rewrite \([x_2 = x_3] (\text{noaliasR } D)\) \(\text{andbT}\)

   Here a rewrite by \(\text{andbT} : \forall b. b \land \text{true} = b\) is used to remove the true left in the goal after rewriting by \(\text{noaliasR}\).

5. Or, we can supply one (or both) of the pointer arguments directly to \(\text{noaliasR}\).

   Goal: \((\!(x_1 \neq x_2) \&\& (x_2 \neq x_3)) = (x_1 \neq x_2)\)
   Proof: by rewrite \((\text{noaliasR } (x := x_2) \land \text{andbT})\)

   Goal: \((\!(x_1 \neq x_2) \&\& (x_2 \neq x_3)) = (x_1 \neq x_2)\)
   Proof: by rewrite \((\text{noaliasR } (y := x_3) \land \text{andbT})\)
7 Related work

Expressive type systems for proof automation. A number of recent languages consider specifying tactics via very expressive dependent types. Examples include VeriML (Stampoulis & Shao, 2010; Stampoulis & Shao, 2012) for automating proofs in \( \lambda \text{HOL} \), and Delphin (Poswolsky & Schürmann, 2009) and Beluga (Pientka & Dunfield, 2008) for proofs in LF. Their starting point is the higher-order abstract syntax (HOAS) style of term representation; consequently, one of their main concerns is using types to track the variable contexts of subgoals generated during tactic execution. In contrast, we do not build a separate language on top of Coq, but rather customize Coq’s unification algorithm. This is much more lightweight, as we do not need to track variable contexts in types, but it also comes with limitations. For example, our automations are pure logic programs, whereas the other proposals may freely use imperative features. On the other hand, as we have demonstrated, canonical structures can benefit from freely mixing with Coq’s primitives for higher-order computation. The mixing would not have been possible had the automation and the base language been kept separated, as is the case in other proposals. Another benefit of the tight integration is that canonical structures can be used to automate not only proofs, but also more general aspects of type inference (e.g., overloading).

In this paper, we have not considered HOAS representations, but we have successfully made first steps in that direction. The interested reader can find in our source files an implementation of the motivating example from VeriML, which considers a simple automation tactic for a logic with quantifiers.

Canonical structures. One important application of canonical structures is described by Bertot et al. (2008), where the ability to key on terms, rather than just types, is used for encoding iterated versions of classes of algebraic operators.

Gonthier (2011) describes a library for matrix algebra in Coq, which introduces a variant of the tagging and lemma selection patterns, but briefly, and as a relatively small part of a larger mathematical development. In contrast, in the current paper, we give a more abstract and detailed presentation of the general tagging pattern and explain its operation with a careful trace. We also present several other novel design patterns for canonical structures, and explore their use in reasoning about heap-manipulating programs.

Asperti et al. (2009) present unification hints, which generalize Coq’s canonical structures by allowing that a canonical solution be declared for any class of unification equations, not only for equations involving a projection out of a structure. Hints are shown to support applications similar to our reflection pattern from Section 4. However, they come with limitations; for example, the authors comment that hints cannot support backtracking. Thus, we believe that the design patterns that we have developed in the current paper are not obviated by the additional generality of hints, and would be useful in that framework as well.

Type classes. Sozeau and Oury (2008) present type classes for Coq, which are similar to canonical structures, but differ in a few important respects. The most salient difference is that inference for type class instances is not performed by unification, but by general proof search. This proof search is triggered after unification, and it is possible to give
a weight to each instance to prioritize the search. This leads to somewhat simpler code, since no tagging is needed, but, on the other hand, it seems less expressive. For instance, we were not able to implement the search-and-replace pattern of Section 5 using Coq type classes, due to the lack of connection between proof search and unification. We were able to derive a different solution for \texttt{bnd\_writeR} using type classes, but the solution was more involved (requiring two specialized classes to differentiate the operations such as \texttt{write} which perform updates to specific heaps, from the operations which merely inspect pointers without performing updates). More importantly, we were not able to scale this solution to more advanced lemmas from our implementation of higher-order separation logic. In contrast, canonical structures did scale, and we provide the overloaded code for these lemmas in our source files (Gonthier et al., 2012).

In the end, we managed to implement all the examples in this paper using Coq type classes, demonstrating that lemma overloading is a useful high-level concept and is not tied specifically to canonical structures. (The implementations using type classes are included in our source files as well; Gonthier et al., 2012.) Nevertheless, unlike for canonical structures, we have not yet arrived at a full understanding of how Coq type classes perform instance resolution. Ultimately, it may turn out that the two formalisms are interchangeable in practice, but we need more experience with type classes to confirm this.

Spitters and van der Weegen (2011) present a reflection algorithm using Coq type classes based on the example of Asperti et al. discussed above. In addition, they consider the use of type classes for overloading and inheritance when defining abstract mathematical structures such as rings and fields. They do not, however, consider lemma overloading more generally as a means of proof automation, as we have presented here.

Finally, in the context of Haskell, Morris and Jones (2010) propose an alternative design for a type class system, called \texttt{ilab}, which is based on the concept of instance chains. Essentially, instance chains avoid the need for overlapping instances by allowing the programmer to control the order in which instances are considered during constraint resolution and to place conditions on when they may be considered. Our tagging pattern (Section 2.3) can be seen as a way of coding up a restricted form of instance chains directly in existing Coq, instead of as a language extension, by relying on knowledge of how the Coq unification algorithm works. \texttt{ilab} also supports failure clauses, which enable one to write instances that can only be applied if some constraint fails to hold. Our approach does not support anything directly analogous, although (as Morris and Jones mention) failure clauses can be encoded to some extent in terms of more heavyweight type class machinery.

**Dependent types modulo theories.** Several recent works have considered enriching the term equality of a dependently typed system to natively admit inference modulo theories. One example is Strub et al.’s CoqMT (Strub, 2010; Barras et al., 2011), which extends Coq’s type checker with first-order equational theories. Another is Jia et al.’s language \(\lambda \cong\) (pronounced “lambda-eek”) (Jia et al., 2010), which can be instantiated with various abstract term-equivalence relations, with the goal of studying how the theoretical properties (e.g., the theory of contextual equivalence) vary with instantiations. Also related are Braibant et al.’s AAC tactics for rewriting modulo associativity and commutativity in Coq (Braibant & Pous, 2010).
In our paper, we do not change the term equality of Coq. Instead, we allow user-supplied algorithms to be executed when desired, rather than by default whenever two terms have to be checked for equality. Moreover, these algorithms do not have to be only decision procedures, but can implement general-purpose computations.

8 Conclusions and future work

The most common approach to proof automation in interactive provers is via tactics, which are powerful but suffer from several practical and theoretical limitations. In this paper, we propose an alternative, specifically for Coq, which we believe puts the problem of interactive proof automation on a stronger foundational footing.

The approach is rooted in the recognition that the type checker and inference engines are already automation tools, and can be coerced via Coq’s canonical structures into executing user-supplied code. Automating proofs in this style is analogous to program overloading via type classes in Haskell. In analogy with the Curry–Howard isomorphism, the automated lemmas are nothing but overloaded programs. In the course of resolving the overloading, the type checker performs the proof automation.

We have illustrated the flexibility and generality of the approach by applying it to a diverse set of lemmas about heaps and pointer aliasing, which naturally arise in verification of stateful programs. Overloading these lemmas required developing a number of design patterns which we used to guide the different aspects of Coq’s unification towards automatically inferring the requisite proofs.

Of course, beyond this, much remains to be done, regarding both the theoretical and pragmatic aspects of our approach. From the theoretical standpoint, we believe it is very important that Coq’s unification algorithm be presented in a formal, declarative fashion, which is currently not the case. In this work we have tried to give a detailed, yet informal, explanation of how it works, with particular focus on the inner workings of canonical instance resolution. But this is far from the complete story.

The study of the unification algorithm is also important from the pragmatic standpoint, as in our experience, the current implementation suffers from a number of peculiar performance problems. For example, we have observed that the time to perform a simple assignment to a unification variable is quadratic in the number of variables in the context, and linear in the size of the term being assigned.

This complexity has so far not been too problematic in practice, as interactive proofs tend to keep variable contexts short, for readability. However, it is a serious concern, and one which is not inherent to overloading or to canonical structures; if addressed by an optimization of Coq’s kernel functionality, it will likely improve many other aspects of the system.

When specifically compared to Ltac, the measures of performance of the cancellation algorithm (our most computation-intensive automation routine) were mixed, varying from version to version of Coq. In some versions, the original Ltac routine from HTT runs faster than the one presented in Section 4, while in others it runs slower. In terms of efficiency, we have yet to develop a clear picture of the tradeoffs between these approaches.
Acknowledgments

We would like to thank Cyril Cohen and Matthieu Sozeau for very helpful discussions, as well as the anonymous reviewers for their thorough and insightful comments. This research has been partially supported by Spanish MINECO projects TIN2010-20639 Paran10 and TIN2012-39391-C04-01 Strongsoft; AMAROUT grant PCOFUND-GA-2008-229599; and Ramon y Cajal grant RYC-2010-0743.

References


