How to Make Ad Hoc Proof Automation Less Ad Hoc

Georges Gonthier
Microsoft Research
gonthier@microsoft.com

Beta Ziliani
MPI-SWS
beta@mpi-sws.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Derek Dreyer
MPI-SWS
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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Logic and constraint programming

General Terms Languages, Design, Theory, Verification

Keywords Interactive theorem proving, custom proof automation, Coq, canonical structures, type classes, tactics, Hoare Type Theory

This research has been partially supported by Spanish MICINN Project TIN2010-20639 Paran10; AMAROUT grant PCOFUND-GA-2008-229599; and Ramon y Cajal grant RYC-2010-0743.

ICFP'11, September 19–21, 2011, Tokyo, Japan.

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 [8] (in Coq), the verification of the optimizing compiler CompCert [17] (also in Coq), and the verification of the operating system microkernel seL4 [16] (in Isabelle). The abovementioned proof assistants employ higher-order logics and type systems in order to maximize expressiveness and generality, but also to facilitate modularity and reuse of verification effort. 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:

$$\begin{aligned}\mathsf{noalias} : \forall h{:}\mathsf{heap}.\, \forall x_1 x_2{:}\mathsf{ptr}.\, \forall v_1{:}A_1.\, \forall v_2{:}A_2.\\ \mathsf{def}\,(x_1 \mapsto v_1 \bullet x_2 \mapsto v_2 \bullet h) \to x_1 \mathrel{!}= x_2\end{aligned}$$

Here, the type heap classifies finite maps from pointers of type 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 def h, declaring that h is not undefined. Consequently, def $(h_1 \bullet h_2)$ holds iff h_1 and h_2 are disjoint heaps. Finally, the conclusion x_1 != x_2 is in fact a *term* of type bool, which Coq implicitly coerces to the *proposition* $(x_1 \mathrel{!}= x_2) = \mathsf{true}$. 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.,*

$$(x_1 \mathrel{!}= x_2)\ \&\&\ (x_2 \mathrel{!}= x_3)\ \&\&\ (x_3 \mathrel{!}= x_1),$$

under the following hypothesis:

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

Before noalias can be applied to prove, say, x_2 != 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:

$$D' : \mathsf{def}\,(x_2 \mapsto v_2 \bullet x_3 \mapsto v_3 \bullet i_1 \bullet i_2 \bullet x_1 \mapsto v_1)$$

Otherwise, the noalias lemma will not apply. Because $\bullet$ 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 != x_2 and x_3 != 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 pro-

vide a separate language for writing *tactics*, which are customized procedures for solving a class of proof obligations. For example, one can write an auto_noalias tactic to solve a goal like x_2 != 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 [6, 7], 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.,*

$$G : \mathsf{if}\ (x_2 == x_3)\ \&\&\ (x_1 != x_2)\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2$$

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

$$\mathsf{negbTE} : \forall b{:}\mathsf{bool}.\, !b = \mathsf{true} \to b = \mathsf{false},$$

thus transforming auto_noalias into a rewrite rule for positive facts of the form $(x_2 == x_3) = \mathsf{false}$. Consequently, we could apply rewrite (negbTE (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.

Our approach involves a sophisticated application of Coq's *canonical structures*, which have existed in Coq for quite some time [23], 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* [28, 13], 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.[1] 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 x_1 and x_2 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 x_2 != x_3, type inference will trigger construction of a canonical instance proving that the heap characterized by D contains bindings for x_2 and x_3 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 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

[1] It is folklore that one can simulate logic programming to some extent using Haskell's multi-parameter classes with functional dependencies [15] or with associated types [5], but Haskell's lack of support for backtracking during type inference limits what kinds of logic programming idioms are possible.

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) [19], where they have replaced often-used tactics. The code in this paper and HTT itself is built on top of Ssreflect [10], 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 at `http://www.mpi-sws.org/~beta/lessadhoc`.

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 from a structure (*i.e.,* type) taken from the standard Ssreflect library [10]:

$$\begin{array}{l}\underline{\mathsf{structure}}\ \mathsf{eqType} := \mathsf{EqType}\ \{\mathsf{sort} : \mathsf{Type};\\ \qquad\qquad\qquad\qquad\quad \mathsf{equal} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{bool};\\ \qquad\qquad\qquad\qquad\quad \_ : \forall x\ y : \mathsf{sort}.\ \mathsf{equal}\ x\ y \leftrightarrow x = y\}\end{array}$$

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

$$\mathsf{eqType\_bool} := \mathsf{EqType}\ \mathsf{bool}\ \mathsf{eq\_bool}\ \mathsf{pf\_bool}$$

where $\mathsf{eq\_bool}\ x\ y := (x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$, and pf_bool is a proof, omitted here, that $\forall x\ y : \mathsf{bool}.\ \mathsf{eq\_bool}\ x\ y \leftrightarrow x = y$.

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

$$\begin{array}{lcl}\mathsf{sort} & : & \mathsf{eqType} \to \mathsf{Type}\\ \mathsf{equal} & : & \forall T{:}\mathsf{eqType}.\ \mathsf{sort}\ T \to \mathsf{sort}\ T \to \mathsf{bool}\end{array}$$

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 $\mathsf{equal}\ x\ y$ instead of $\mathsf{equal}\ T\ x\ y$, 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 $A \times B$, where A and B are themselves instances of eqType:

$$\begin{array}{l}\mathsf{eqType\_pair}\ (A\ B : \mathsf{eqType}) :=\\ \qquad \mathsf{EqType}\ (\mathsf{sort}\ A \times \mathsf{sort}\ B)\ (\mathsf{eq\_pair}\ A\ B)\ (\mathsf{pf\_pair}\ A\ B)\end{array}$$

where

$$\begin{array}{l}\mathsf{eq\_pair}\ (A\ B : \mathsf{eqType})\ (u\ v : \mathsf{sort}\ A \times \mathsf{sort}\ B) :=\\ \qquad \mathsf{equal}\ (\pi_1\ u)\ (\pi_1\ v)\ \&\&\ \mathsf{equal}\ (\pi_2\ u)\ (\pi_2\ v)\end{array}$$

and pf_pair is omitted as before.

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 $\mathsf{equal}\ (b_1, b_2)\ (c_1, c_2)$, where b_1, b_2, c_1 and c_2 are of type bool, it will generate a unification problem of the form

$$\mathsf{sort}\ ?T\ \widehat{=}\ \mathsf{bool} \times \mathsf{bool}$$

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 $?B$:

$$\mathsf{sort}\ ?A\ \widehat{=}\ \mathsf{bool} \qquad \mathsf{sort}\ ?B\ \widehat{=}\ \mathsf{bool}$$

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

In this manner, canonical instances can be used for *overloading*, similar to the way type classes are used in Haskell [28, 13].[2] 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 $\mathsf{equal}\ x\ y$, and the typechecker 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 typechecker that *for each* projection proj of the structure S, and c the *head symbol* of $\mathsf{proj}\ V$, the unknown X in the unification equation

$$\mathsf{proj}\ X\ \widehat{=}\ c\ x_1 \ldots x_n \qquad (*)$$

should by default be solved by unifying $X\ \widehat{=}\ V$. For instance, in the case of eqType_pair, 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. We emphasize that: (1) to control the number of such default facts, we will frequently anonymize the projections if they are not important for the application, as in the case of the proof in eqType above; and (2) there can only be one specified default solution for any given proj and c (*i.e.,* overlapping canonical instances 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 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.

[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.

To make matters concrete, let's 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 a heap h. If the search is successful, that is, if h is of the form

$$\cdots \bullet (\cdots \bullet x \mapsto v \bullet \cdots) \bullet \cdots ,$$

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

$$\underline{\mathsf{structure}}\ \mathsf{find}\ x := \mathsf{Find}\ \{\mathsf{heap\_of} : \mathsf{heap};\ \_ : \mathsf{invariant}\ x\ \mathsf{heap\_of}\}$$

where invariant is defined as

$$\mathsf{invariant}\ x\ h := \mathsf{def}\ h \to x \in \mathsf{dom}\ h$$

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 $x \in \mathsf{dom}\ \mathsf{heap\_of}$ (under the assumption that heap_of is well-defined).

The search task will commence when some input heap h gets unified with $\mathsf{heap\_of}\ X$ for an unknown $X : \mathsf{find}\ x$, at which point Coq's unification algorithm will recursively deconstruct h in order to search for a canonical implementation of X such that $\mathsf{heap\_of}\ X = h$. If that search is successful, the last field of X will be a proof of $\mathsf{invariant}\ x\ h$, which we can apply to a proof of $\mathsf{def}\ h$ to obtain a proof of $x \in \mathsf{dom}\ h$, as desired. (By way of analogy, this is similar to what we previously used for eqType_pair. The construction of a canonical equality operator at a given type A will commence when A is unified with $\mathsf{sort}\ T$ for an unknown $T : \mathsf{eqType}$, 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 actually implement the recursion:

$$\underline{\mathsf{canonical}}\ \mathsf{found\_struct}\ A\ x\ (v : A) :=\ \mathsf{Find}\ x\ (x \mapsto v)\ (\mathsf{found\_pf}\ A\ x\ v)$$

$$\underline{\mathsf{canonical}}\ \mathsf{left\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=\ \mathsf{Find}\ x\ ((\mathsf{heap\_of}\ f) \bullet h)\ (\mathsf{left\_pf}\ x\ h\ f)$$

$$\underline{\mathsf{canonical}}\ \mathsf{right\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=\ \mathsf{Find}\ x\ (h \bullet (\mathsf{heap\_of}\ f))\ (\mathsf{right\_pf}\ x\ h\ f)$$

Note that the first argument to the constructor Find in these instances is the parameter x of the find structure.

The first instance, found_struct, corresponds to the case where the heap_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, left_struct and right_struct, handle the cases where the heap_of projection is of the form $h_1 \bullet h_2$, 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 $f : \mathsf{find}\ x$ whose heap_of projections are unified with the subheaps h_1 or h_2 of the original heap_of projection.

Notational Convention 2. In the declarations above, found_pf, left_pf and right_pf are proofs, witnessing that invariant 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: $\mathsf{Find}\ x\ ((\mathsf{heap\_of}\ f) \bullet h)\ \ldots$

Unfortunately, this set of canonical instances does not quite work. The trouble is that left_struct and right_struct are overlapping instances since both match against the same head symbol (namely, •), 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.

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 diverges significantly from how unification works in, say, Haskell. We will thus first illustrate the pattern concretely in terms of our find example, and then explain afterwards how it solves the problem.

The Pattern First, we define a "tagged" version of the type of thing we're recursively analyzing—in this case, the heap type:

$$\underline{\mathsf{structure}}\ \mathsf{tagged\_heap} := \mathsf{Tag}\ \{\mathsf{untag} : \mathsf{heap}\}$$

This structure declaration also introduces two functions witnessing the isomorphism between heap and tagged_heap:

$$\begin{array}{rcl} \mathsf{Tag} & : & \mathsf{heap} \to \mathsf{tagged\_heap} \\ \mathsf{untag} & : & \mathsf{tagged\_heap} \to \mathsf{heap} \end{array}$$

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

$$\mathsf{invariant}\ x\ (h : \mathsf{tagged\_heap}) :=\ \mathsf{def}\ (\mathsf{untag}\ h) \to x \in \mathsf{dom}\ (\mathsf{untag}\ h)$$

$$\underline{\mathsf{structure}}\ \mathsf{find}\ x := \mathsf{Find}\ \{\mathsf{heap\_of} : \mathsf{tagged\_heap};\ \_ : \mathsf{invariant}\ x\ \mathsf{heap\_of}\}$$

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 doesn't matter much in this particular example, but it does in other examples in the paper.)

$$\begin{array}{l} \mathsf{right\_tag}\ h := \mathsf{Tag}\ h \\ \mathsf{left\_tag}\ h := \mathsf{right\_tag}\ h \\ \underline{\mathsf{canonical}}\ \mathsf{found\_tag}\ h := \mathsf{left\_tag}\ h \end{array}$$

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

$$\underline{\mathsf{canonical}}\ \mathsf{found\_struct}\ A\ x\ (v : A) :=\ \mathsf{Find}\ x\ (\mathsf{found\_tag}\ (x \mapsto v))\ \ldots$$

$$\underline{\mathsf{canonical}}\ \mathsf{left\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=\ \mathsf{Find}\ x\ (\mathsf{left\_tag}\ ((\mathsf{untag}\ (\mathsf{heap\_of}\ f)) \bullet h))\ \ldots$$

$$\underline{\mathsf{canonical}}\ \mathsf{right\_struct}\ x\ h\ (f : \mathsf{find}\ x)\ :=\ \mathsf{Find}\ x\ (\mathsf{right\_tag}\ (h \bullet (\mathsf{untag}\ (\mathsf{heap\_of}\ f))))\ \ldots$$

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 unification—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 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

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f)\ \widehat{=}\ h,$$

for some unknown $?f : \mathsf{find}\ x$. Since found_tag is canonical, the problem will be reduced to unifying

$$\mathsf{heap\_of}\ ?f\ \widehat{=}\ \mathsf{found\_tag}\ h.$$

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

$$\mathsf{heap\_of}\ ?f\ \widehat{=}\ \mathsf{left\_tag}\ h,$$

which will in turn cause Coq to try unifying $?f$ with some instantiation of left_struct. Again, if that fails, 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 *un*-overloaded version:

$$\begin{array}{lcl}\mathsf{indom} & : & \forall x{:}\mathsf{ptr}.\,\forall v{:}A.\,\forall h{:}\mathsf{heap}.\\ & & \quad \mathsf{def}\ (x \mapsto v \bullet h) \to x \in \mathsf{dom}\ (x \mapsto v \bullet h)\end{array}$$

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:

$$\begin{array}{lcl}\mathsf{indomR} & : & \forall x{:}\mathsf{ptr}.\,\forall f{:}\mathsf{find}\ x.\\ & & \quad \mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f)) \to\\ & & \quad\quad x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f))\end{array}$$

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

To demonstrate the automated nature of indomR on a concrete example, we will walk in detail through the "trace" of Coq type inference when indomR is applied to prove the goal

$$z \in \mathsf{dom}\ h$$

in a context where $x\ y\ z : \mathsf{ptr}$, $u\ v\ w : A$, $h : \mathsf{heap} := x \mapsto u \bullet y \mapsto v \bullet z \mapsto w$, and $D : \mathsf{def}\ h$. When indomR is applied, its formal parameters are turned into unification variables $?x$ and $?f : \mathsf{find}\ ?x$, which will be constrained by the unification process. (Hereafter, we will use $?$ to denote unification variables, with previously unused $?x$'s denoting fresh unification variables.)

As a first step, the system tries to unify

$$?x \in \mathsf{dom}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ ?f))\ \widehat{=}\ z \in \mathsf{dom}\ h$$

getting $?x = z$, and, then

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f)\ \widehat{=}\ h$$

By canonicity of found_tag, we need to solve

$$\mathsf{heap\_of}\ ?f\ \widehat{=}\ \mathsf{found\_tag}\ h$$

Unification tries to instantiate $?f$ with 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 found_tag, yielding, as $\bullet$ is left-associative,

$$\mathsf{heap\_of}\ ?f\ \widehat{=}\ \mathsf{left\_tag}\ ((x \mapsto u \bullet y \mapsto v) \bullet z \mapsto w) \quad (1)$$

Now, left_struct can be used for $?f$, and the following unification problems eventually arise:

$$\begin{array}{rcl}\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2) & \widehat{=} & x \mapsto u \bullet y \mapsto v\\ ?h & \widehat{=} & z \mapsto w\\ ?f & \widehat{=} & \mathsf{left\_struct}\ z\ ?h\ ?f_2\end{array}$$

Attempting to solve the first equation, the system again applies found_tag and found_struct. As before, it fails, unfolds found_tag to get left_tag, then attempts left_struct to get

$$\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_3)\ \widehat{=}\ x \mapsto u,$$

It can now apply found_tag and found_struct to finally fail, because z does not match x. Rolling back, it unfolds left_tag to right_tag, matches z with y, fails again, and rolls back to the equation (1). At this point it unfolds left_tag to right_tag, resulting in

$$\mathsf{heap\_of}\ ?f\ \widehat{=}\ \mathsf{right\_tag}\ (x \mapsto u \bullet y \mapsto v \bullet z \mapsto w),$$

then chooses right_struct to eventually obtain

$$\begin{array}{rcl}?h' & \widehat{=} & x \mapsto u \bullet y \mapsto v\\ \mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2') & \widehat{=} & z \mapsto w\\ ?f & \widehat{=} & \mathsf{right\_struct}\ z\ ?h'\ ?f_2'\end{array}$$

The first equation unifies immediately, after which the second one is solved by applying found_tag and choosing found_struct for $?f_2'$. After that, the third equation also unifies right away.

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

$$x \mapsto v_1 \bullet (h_3 \bullet h_4) = h_4 \bullet x \mapsto v_2$$

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

$$v_1 = v_2 \wedge h_3 = \mathsf{empty}$$

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 $\bullet$ 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.

Cancellation Since the second stage is simpler, we explain it first. For the purposes of presentation, we restrict our pointers to only store values of some predetermined type T (although the actual implementation in our source files is more general). The data type that we use for syntactic representation of heap expressions is then the following:

$$\begin{aligned}\mathsf{elem} &:= \mathsf{Var\ of\ nat} \mid \mathsf{Pts\ of\ nat}\ \&\ T\\ \mathsf{term} &:= \mathsf{seq\ elem}\end{aligned}$$

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 a number identifying the *heap* variable in the style of de Bruijn indices; that is, as an index into some environment (to be explained below). In the second case, the component is represented as Pts $m\ v$, where m is a number 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 $\bullet$ is commutative and associative, lists suffice for our purposes.

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

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

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. 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 hlook and plook 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:

```
cancel (i : ctx) (t1 t2 r : term) : Prop :=
  match t1 with
   nil ⇒ interp i r = interp i t2
  | Pts m v :: t1' ⇒
     if premove m t2 is Some (v', t2') then
       cancel i t1' t2' r ∧ v = v'
     else cancel i t1' t2 (Pts m v :: r)
  | Var n :: t1' ⇒
     if hremove n t2 is Some t2' then cancel i t1' t2' r
     else cancel i t1' t2 (Var n :: r)
  end
```

Figure 1. Heap cancellation algorithm.

```
interp (i : ctx) (t : term) : heap :=
  match t with
   Var n :: t' ⇒ if hlook i n is Some h then  h • interp i t'
                 else Undef
  | Pts m v :: t' ⇒
      if plook i m is Some x then x ↦ v • interp i t'
      else Undef
  | nil ⇒ empty
  end
```

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

$$\begin{aligned}\mathsf{interp}\ i\ [\mathsf{Pts}\ 0\ v_1, \mathsf{Var}\ 0, \mathsf{Var}\ 1] &= x \mapsto v_1 \bullet (h_3 \bullet (h_4 \bullet \mathsf{empty}))\\ \mathsf{interp}\ i\ [\mathsf{Var}\ 1, \mathsf{Pts}\ 0\ v_2] &= h_4 \bullet (x \mapsto v_2 \bullet \mathsf{empty})\end{aligned}$$

Given this definition of term, we can now encode the cancellation algorithm as a predicate (*i.e.,* a function into 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 deBruijn 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 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 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, *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 cancel immediately returns it (our actual implementation performs some additional optimization before returning). The helper function 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, hremove $n\ t_2$ searches for Var n in t_2 and if found, returns t_2' obtained from t_2 after removal of n.

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

```
cancel_sound : ∀i : ctx. ∀t1 t2 : term.
  def (interp i t1) → interp i t1 = interp i t2 →
  cancel i t1 t2 nil.
```

The proof of cancel_sound is rather involved so we omit it here, but it can be found 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 above (see for example the work of Grégoire et al. [12]). Where we differ from the related work is 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.

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 equation sides, 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 $\mathsf{union\_tag}(f_1 \bullet f_2)$ instead of $\mathsf{union\_tag}((\mathsf{untag}\ (\mathsf{heap\_of}\ f_1)) \bullet (\mathsf{untag}\ (\mathsf{heap\_of}\ f_2)))$, 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:

$$\mathsf{interp}\ j\ t = \mathsf{heap\_of}$$

There are additional invariants too, which are necessary to carry out the proofs, but we omit them here for brevity; they can be found in our 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 (as usual, reverse) ordering of tag synonyms. 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.

When reflecting the empty heap, the instance is empty_struct. In this case, the input context i is simply passed 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 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_2, and returned reflected elem is again Pts m v. Similar

```
var_tag h := Tag h
pts_tag h := var_tag h
empty_tag h := pts_tag h
canonical union_tag h := empty_tag h

structure ast (i j : ctx) (t : term) :=
  Ast {heap_of : tagged_heap;
       _ : interp j t = heap_of ∧ ...}

canonical union_struct (i j k : ctx) (t1 t2 : term)
                       (f1 : ast i j t1)(f2 : ast j k t2) :=
  Ast i k (append t1 t2) (union_tag (f1 • f2)) ...

canonical empty_struct (i : ctx) :=
  Ast i i nil (empty_tag empty) ...

canonical pts_struct (hs : seq heap) (xs1 xs2 : seq ptr)
                     (m : nat) (v : A) (f : xfind xs1 xs2 m) :=
  Ast (hs, xs1) (hs, xs2) [Pts m v] (pts_tag (f ↦ v)) ...

canonical var_struct (hs1 hs2 : seq heap) (xs : seq ptr)
                     (n : nat) (f : xfind hs1 hs2 n) :=
  Ast (hs1, xs) (hs2, xs) [Var n] (var_tag f) ...
```

Figure 2. Structure ast for reflecting a heap.

```
structure xtagged A := XTag {xuntag : A}

extend_tag  A (x : A) := XTag  x
recurse_tag A (x : A) := extend_tag x
canonical found_tag A (x : A) := recurse_tag  x

structure xfind A (s r : seq A) (i : nat) :=
  XFind {elem_of : xtagged A;
         _ : index r i = elem_of ∧ ...}

canonical found_struct A (x : A) (s : seq A) :=
  XFind (x :: s) (x :: s) 0 (found_tag x) ...

canonical recurse_struct (i : nat) (y : A) (s r : seq A)
                         (f : xfind s r i) :=
  XFind (y :: s) (y :: r) (i + 1) (recurse_tag f) ...

canonical extend_struct A (x : A) :=
  XFind nil [x] 0 (extend_tag x) ...
```

Figure 3. Structure xfind for searching for an element in a list; appending the element at the end if not found.

considerations apply in the case when 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 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 not 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.

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 xs_1 and hs_1 in the instances pts_struct and var_struct, respectively. As we will argue in Section 6, such unification leads to incorrect results, if done directly, but we will be able to perform it *indirectly*, using a new design pattern.

Now we can present the overloaded lemma cancelR.

$$\begin{array}{l}\mathsf{cancelR} : \forall j\ k : \mathsf{ctx}.\, \forall t_1\ t_2 : \mathsf{term}.\\ \qquad \forall f_1 : \mathsf{ast\ nil}\ j\ t_1.\, \forall f_2 : \mathsf{ast}\ j\ k\ t_2.\\ \quad \mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f_1)) \to\\ \quad \mathsf{untag}\ (\mathsf{heap\_of}\ f_1) = \mathsf{untag}\ (\mathsf{heap\_of}\ f_2) \to\\ \quad \mathsf{cancel}\ k\ t_1\ t_2\ \mathsf{nil}\end{array}$$

Assuming we have a hypothesis

$$H : \overbrace{x \mapsto v_1 \bullet (h_3 \bullet h_4)}^{h_1} = \overbrace{h_4 \bullet x \mapsto v_2}^{h_2}$$

and a hypothesis $D : \mathsf{def}\ \ h_1$, we can apply

$$\mathsf{move}/(\mathsf{cancelR}\ D) : H.$$

This will make Coq fire the following unification problems:

1. $\mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_1)) \ \widehat{=}\ \mathsf{def}\ h_1$
2. $\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_1)) \ \widehat{=}\ h_1$
3. $\mathsf{untag}\ (\mathsf{heap\_of}\ ?f_2)) \ \widehat{=}\ h_2$

Because f_1 and f_2 are variable instances of the structure 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 $v_1 = v_2 \wedge h_3 = \mathsf{empty} \wedge \mathsf{empty} = \mathsf{empty}$. The trailing $\mathsf{empty} = \mathsf{empty}$ can ultimately be removed with a few simple optimizations of cancel, which we omitted to simplify the presentation.

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 "search and replace" functionality, and we describe it in the context of our higher-order implementation of separation logic [22] in Coq. Interestingly, this *search-and-replace* pattern can also be described as higher-order, as it crucially relies on the typechecker'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

$$\mathsf{verify} : \mathsf{prog}\ A \to \mathsf{heap} \to (A \to \mathsf{heap} \to \mathsf{Prop}) \to \mathsf{Prop}.$$

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

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

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 $x : \mathsf{ptr}$ and $v : A$, the program $\mathsf{write}\ x\ v : \mathsf{prog\ unit}$ stores v into x and terminates. We also require the operation for sequential composition, which takes the form of monadic bind:

$$\mathsf{bind} : \mathsf{prog}\ A \to (A \to \mathsf{prog}\ B) \to \mathsf{prog}\ B.$$

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

$$\begin{array}{l}\mathsf{bnd\_write} : \mathsf{verify}\ (e\ ())\ (x \mapsto v \bullet h)\ q \to\\ \qquad \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ (x \mapsto w \bullet h)\ q\end{array}$$

To verify $\mathsf{write}\ x\ v$ 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, bnd_write suffers from the same problem as indom and noalias, as each application requires bringing the pointer x to the top of the heap. We would like to devise an automated version bnd_writeR, but, unlike indomR, application of bnd_writeR cannot merely check if a pointer x is in the heap. It needs to remember the heap from the goal, and reproduce it in the premise, only with the contents of x changed from w to v.

For example, applying bnd_writeR to the goal

$$\begin{array}{l}G_1 : \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x_2\ 4)\ e)\\ \qquad (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2) \bullet (i_2 \bullet x_3 \mapsto 3))\\ \qquad q\end{array}$$

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

$$G_2 : \mathsf{verify}\ (e\ ())\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4) \bullet (i_2 \bullet x_3 \mapsto 3))\ q.$$

The Pattern Here is where functions come in. The 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 $f\ (x \mapsto w)$), obtains 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 $x \mapsto 2$, while in G_2, we want to fill it with $x \mapsto 4$. Hence, in this case, the inferred function f should, intuitively, be:

$$\mathsf{fun}\ k.\, i_1 \bullet (x_1 \mapsto 1 \bullet k) \bullet (i_2 \bullet x_3 \mapsto 3).$$

The function f therefore takes an input heap k, but it should not merely produce an output heap. Instead, we want f's range to be a structure, called here $\mathsf{partition}\ k\ r$ (Figure 4). A projection heap_of out of this structure will be used to trigger the search for the subheap that should be replaced with a hole.

For technical reasons, partition has an additional heap parameter r, whose role will be explained later. Because the range of f depends on the input k, f must have a *dependent function type*, and the bnd_writeR lemma looks as below. For clarity, we use Π to distinguish a function, even when Coq does not make that distinction.

$$\begin{array}{l}\mathsf{bnd\_writeR} : \forall r : \mathsf{heap}.\, \forall f : (\Pi k : \mathsf{heap}.\, \mathsf{partition}\ k\ r).\\ \quad \mathsf{verify}\ (e\ ())\ (f\ (x \mapsto v))\ q \to\\ \quad \mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x\ v)\ e)\ (f\ (x \mapsto w))\ q\end{array}$$

We have omitted the projections and written $f\ (x \mapsto w)$ instead of $\mathsf{untag}\ (\mathsf{heap\_of}\ (f\ (x \mapsto w)))$, and similarly in the case of $x \mapsto v$.

When the bnd_writeR lemma is applied to a verify goal with a heap h, the type checker attempts to unify

$$\mathsf{untag}\ (\mathsf{heap\_of}\ (?f\ (x \mapsto w))) \ \widehat{=}\ h.$$

```
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 : tagged_heap;
             _ : heap_of = k • r}

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

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

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

Figure 4. Structure partition for partitioning a heap into two parts: the part matching k, and "the rest" (r).

The instances in Figure 4 are designed so that the canonical solution f will have 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 simple. The last step in solving the above unification goal will be to unify $?f\ (x \mapsto w)$ with whatever canonical structure is ultimately computed. For example, if $h = h_0 \bullet x \mapsto w$, then the last step of unification will resolve the following goal:

$$?f\ (x \mapsto w) \ \widehat{=}\ \mathsf{right\_struct}\ h_0\ \mathsf{empty}\ \mathsf{found\_struct}\ (x \mapsto w).$$

Coq's unification will reduce to subgoals involving the structural components of the applications:

$$?f \ \widehat{=}\ \mathsf{right\_struct}\ h_0\ \mathsf{empty}\ \mathsf{found\_struct}$$

and

$$x \mapsto w \ \widehat{=}\ x \mapsto w,$$

which are solved immediately. However, this only works because the k parameter of right_struct comes last, thus making it possible to structurally match the occurrences of $x \mapsto w$ on the two sides of the equation. If k came earlier, the unification would simply fail.

We have described how to construct the canonical solution f, but the mere construction is not sufficient to carry out the proof of 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 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 doesn't 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 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, notice that the premise and conclusion of bnd_writeR both contain projections out of f. Therefore, the lemma may be used both in forward reasoning (out of hypotheses) and in backward reasoning (for discharging a given goal). For example, we can prove the goal

$$\mathsf{verify}\ (\mathsf{bind}\ (\mathsf{write}\ x_2\ 4)\ e)\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 2))\ q$$

under hypothesis

$$H\ :\ \mathsf{verify}\ (e\ ())\ (i_1 \bullet (x_1 \mapsto 1 \bullet x_2 \mapsto 4))\ q$$

in two ways:

- *Backward*: By applying bnd_writeR to the goal. The goal will therefore be changed to exactly match H.
- *Forward*: By applying 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 gleaned 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 bnd_writeR that is usable in both directions.

6. Reordering Unification Subproblems

In this section we design an overloaded version of our original noalias lemma from Section 1. The main challenge, as it turns out, is ensuring 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 naïve implementation will result in the triggering of a search for a pointer in a heap before we know what pointer we're searching for. Fortunately, it is possible to handle this problem very easily using an extremely simple design pattern we call *hoisting*.

Before we come to explain the details of the problem and the pattern, let us first present the search structures that form the core of the automation for noalias and are shown in Figures 5–7. 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.

Step (1) is implemented by the scan structure in Figure 5. 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 is 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

```
structure tagged_heap := Tag {untag : heap}
default_tag (h : heap) := Tag h
ptr_tag h := default_tag h
canonical union_tag h := ptr_tag h

structure scan (s : seq ptr) :=
  Scan {heap_of : tagged_heap;
        _ : def heap_of →
              uniq s ∧ ∀x. x ∈ s → x ∈ dom heap_of}

canonical union_struct s1 s2 (f1 : scan s1) (f2 : scan s2) :=
  Scan (append s1 s2) (union_tag (f1 • f2)) ...

canonical ptr_struct A x (v : A) :=
  Scan (x :: nil) (ptr_tag (x ↦ v)) ...

canonical default_struct h := Scan nil (default_tag h) ...
```

Figure 5. Structure scan for computing a list of pointers syntactically appearing in a heap.

```
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)) ...
```

Figure 6. 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) :=
  Seach1 x (recurse_tag (y :: f)) ...
```

Figure 7. Structure search1 for finding a pointer in a list.

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.

With our core automated machinery in hand, we are ready for our first (failed) attempt at an overloaded version of noalias:

```
noaliasR_wrong :
  ∀x : ptr. ∀s : seq ptr. ∀f : scan s. ∀g : check x s.
    def (untag (heap_of f)) → x != unpack (y_of g)
```

The intuition behind the reformulation can be explained in programming terms. When the lemma is applied to a hypothesis D of type def h, the heap h will be unified with the projection untag (heap_of f). This will trigger an inference problem in which the system solves for the canonical implementation of f by executing the scan algorithm, thus producing as output the pointer list s. For example, if

$$h = i_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (i_2 \bullet x_3 \mapsto v_3),$$

then s will get unified with $[x_1, x_2, x_3]$, so it can serve as a well-defined input to the search steps that follow.

When the lemma is subsequently applied in order to solve a goal of the form $x' \texttt{ != } y'$, we need some way to get the unification with the conclusion of the lemma to trigger the automated search for x' and y' in the list s. Toward this end, we can use the unification with either x' or y' as the trigger for the search, and here we choose the latter. Specifically, we define a structure check, whose construction is keyed on a projection y_of that will be unified with y':[3]

```
structure packed_ptr := Pack {unpack : ptr}

canonical pack (z : ptr) := Pack z

structure check (x : ptr) (s : seq ptr) :=
  Check {y_of : packed_ptr;
         _ : uniq s → x != unpack y_of}

canonical start x y (s2 : search2 x y) :=
  Check x (untag2 (seq2_of s2)) (pack y) ...
```

The sole purpose of the canonical instance start for the check structure is to take x' and s, passed in as parameters, and y', passed in through unification with the y_of projection, and repackage them appropriately in the form that the search2 structure expects. In particular, recall that search2 expects the two pointers to be passed in as parameters, and s to be unified with its seq2_of projection.

Unfortunately, the linking structure we've defined here doesn't quite work. The trouble has to do with the way in which Coq orders the subproblems that arise during canonical instance unification. Although a fully detailed presentation of the Coq unification algorithm is beyond the scope of this paper, the rule of thumb is that when matching against a canonical instance, Coq solves the unification subproblems in a *left-to-right* order—that is, it first solves the unification subproblems corresponding to each of the structure parameters (in the case of check, the x and s parameters) and only then unifies the structure projections (like y_of in this case).

Thus, when matching against the start instance, what happens is the following. First, fresh unification variables are generated for the instance parameters $?x$, $?y$ and $?s_2$. Then, three new unification subproblems are generated and solved in the following order,

[3] The astute reader may notice that it is not actually necessary to tag (or in this case, pack) the y_of projection of the check structure since we are only defining one canonical instance for the structure. We tag y_of simply to minimize the delta required when we describe the hoisting pattern below.

corresponding to the arguments of Check:

$$?x \;\widehat{=}\; x'$$
$$\mathsf{untag2}(\mathsf{seq2\_of}\ ?s_2) \;\widehat{=}\; s$$
$$?y \;\widehat{=}\; y'$$

The problem is that the solution to the second equation will fire the search for a canonical solution for $?s_2$ of type search2 x' $?y$—thus triggering the search for x' and $?y$ in s—*before* the third equation has unified $?y$ with y'. So we will end up searching s for an unknown $?y$, leading to the wrong behavior in most cases.

The Pattern In order to fix our check structure, we need a way to arrange for $?y$ to be unified with y' before the search algorithm gets triggered on the pointer list s. The trick to doing this is to give check an extra parameter y, which will appear earlier than (*i.e.,* to the left of) s in the parameter list, thus ensuring higher priority in the unification order. But we will also constrain that parameter y to be equal to the y_of projection from the structure by (effectively) giving the projection a singleton type. We call this *hoisting*.

To illustrate, here is how to change the packed_ptr and check structures (and their instances) according to the hoisting pattern:

$$\underline{\mathsf{structure}}\ \mathsf{equals\_ptr}\ (z : \mathsf{ptr}) := \mathsf{Pack}\ \{\mathsf{unpack} : \mathsf{ptr}\}$$

$$\underline{\mathsf{canonical}}\ \mathsf{equate}\ (z : \mathsf{ptr}) := \mathsf{Pack}\ z\ z$$

$$\begin{array}{l}\underline{\mathsf{structure}}\ \mathsf{check}\ (x\ y : \mathsf{ptr})\ (s : \mathsf{seq\ ptr}) := \\ \quad \mathsf{Check}\ \{\mathsf{y\_of} : \mathsf{equals\_ptr}\ y; \\ \qquad\qquad \_ : \mathsf{uniq}\ s \to x\ \mathsf{!=}\ \mathsf{unpack\ y\_of}\}\end{array}$$

$$\begin{array}{l}\underline{\mathsf{canonical}}\ \mathsf{start}\ x\ y\ (s_2 : \mathsf{search2}\ x\ y) := \\ \quad \mathsf{Check}\ x\ y\ (\mathsf{untag2}\ (\mathsf{seq2\_of}\ s_2))\ (\mathsf{equate}\ y)\ \ldots\end{array}$$

The key here is the new version of packed_ptr, which (for clarity) we call equals_ptr, and which is now explicitly parameterized over a pointer z. The instance equate guarantees that the canonical value of type equals_ptr z is a package containing z itself. We rely on this guarantee in the check structure, whose y_of projection now has type equals_ptr y, thus constraining it to be equal to check's new parameter y.

We can now revise our statement of the overloaded noaliasR lemma ever so slightly to mention the new parameter y:

$$\begin{array}{l}\mathsf{noaliasR}: \\ \quad \forall x\ y : \mathsf{ptr}.\ \forall s : \mathsf{seq\ ptr}.\ \forall f : \mathsf{scan}\ s.\ \forall g : \mathsf{check}\ x\ y\ s. \\ \qquad \mathsf{def}\ (\mathsf{untag}\ (\mathsf{heap\_of}\ f)) \to x\ \mathsf{!=}\ \mathsf{unpack}\ (\mathsf{y\_of}\ g)\end{array}$$

As above, 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 x' != y', the unification engine will unify x' with the argument $?x$ of noaliasR, and proceed to unify

$$\mathsf{unpack}\ (\mathsf{y\_of}\ ?g) \;\widehat{=}\; y'$$

in a context where $?g$: check x' $?y$ s. Note that, although we have elided it, unpack here really takes as its first (implicit) argument the unknown pointer $?y$ that is implied by equals_ptr $?y$ (the type of y_of $?g$). Thus, by canonicity, the equation will be resolved with equate, and two new equations will be generated (in this order):

$$?y \;\widehat{=}\; y'$$
$$\mathsf{y\_of}\ ?g \;\widehat{=}\; \mathsf{equate}\ y'$$

The first unification is the essential one that needs to happen prior to triggering of the search procedure, so that all inputs to the search are known; the rest of unification then works as expected.

Intuitively, the reason the equation between $?y$ and y' is generated first is because it arises from unifying the *types* of y_of $?g$ and equate y', which is necessary before one can unify the terms themselves. For a more formal explanation of canonical instance resolution, we refer the reader to our online appendix [11].

Finally, note the y that is shared between the parameter (y) and the projection (equate y) of the start instance. By hoisting y so that it appears before s in the argument list of Check, we have ensured that it will be unified with the y' from the goal before the search through s begins.

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

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

Assume further that the arguments x, y, s, f and g of noaliasR are implicit, so that we can write simply (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.

$$\begin{array}{l}\mathrm{Goal}\ :\ x_1\ \mathsf{!=}\ x_2 \\ \mathrm{Proof}:\ \mathsf{by\ apply}:\ (\mathsf{noaliasR}\ D).\end{array}$$

2. The lemma can be used in iterated rewriting (notice the modifier "!"). 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.

$$\begin{array}{l}\mathrm{Goal}\ :\ (x_1\ \mathsf{!=}\ x_2)\ \&\&\ (x_2\ \mathsf{!=}\ x_3)\ \&\&\ (x_3\ \mathsf{!=}\ x_1) \\ \mathrm{Proof}:\ \mathsf{by\ rewrite\ !}(\mathsf{noaliasR}\ D).\end{array}$$

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 : $\forall b$:bool. $!b = \mathsf{true} \to b = \mathsf{false}$, we have:

$$\begin{array}{l}\mathrm{Goal}\ :\ \mathsf{if}\ (x_2\ \mathsf{==}\ x_3)\ \&\&\ (x_1\ \mathsf{!=}\ x_2)\ \mathsf{then\ false\ else\ true} \\ \mathrm{Proof}:\ \mathsf{by\ rewrite}\ (\mathsf{negbTE}\ \ (\mathsf{noaliasR}\ D)).\end{array}$$

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.

$$\begin{array}{l}\mathrm{Goal}\ :\ ((x_1\ \mathsf{!=}\ x_2)\ \&\&\ (x_2\ \mathsf{!=}\ x_3)) = (x_1\ \mathsf{!=}\ x_2) \\ \mathrm{Proof}:\ \mathsf{by\ rewrite}\ [x_2\ \mathsf{!=}\ x_3](\mathsf{noaliasR}\ D)\ \mathsf{andbT}.\end{array}$$

   Here a rewrite by andbT : $\forall b.\ b\ \&\&\ \mathsf{true} = b$ is used to remove the true left in the goal after rewriting by noaliasR.

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

$$\begin{array}{l}\mathrm{Goal}\ :\ ((x_1\ \mathsf{!=}\ x_2)\ \&\&\ (x_2\ \mathsf{!=}\ x_3)) = (x_1\ \mathsf{!=}\ x_2) \\ \mathrm{Proof}:\ \mathsf{by\ rewrite}\ (\mathsf{noaliasR}\ (x := x_2)\ D)\ \mathsf{andbT}.\end{array}$$

$$\begin{array}{l}\mathrm{Goal}\ :\ ((x_1\ \mathsf{!=}\ x_2)\ \&\&\ (x_2\ \mathsf{!=}\ x_3)) = (x_1\ \mathsf{!=}\ x_2) \\ \mathrm{Proof}:\ \mathsf{by\ rewrite}\ (\mathsf{noaliasR}\ (y := x_3)\ D)\ \mathsf{andbT}.\end{array}$$

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 [26] for automating proofs in Coq, and Delphin [21] and Beluga [20] for proofs in Twelf. 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. [3], where the ability to key on terms, rather than just types, is used for encoding iterated versions of classes of algebraic operators.

Gonthier [9] describes a library for matrix algebra in Coq, which introduces a variant of the tagging pattern, 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. [1] 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 [24] 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 nor hoisting 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 bnd_writeR using type classes, but the solution was more involved (requiring two specialized classes to differentiate the operations such as 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 [11].

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 [11].) Nevertheless, unlike for canonical structures, we have not yet arrived at a full understanding of how Coq type classes perform instance resolution. In addition, the preliminary performance results are mixed, with type classes sometimes beating canonical structures (in terms of speed) and sometimes vice versa. 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 [25] 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 [18] propose an alternative design for a type class system, called `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. `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 [27, 2], which extends Coq's typechecker with *first-order equational* theories. Another is Jia et al.'s language $\lambda^{\cong}$ (pronounced "lambda-eek") [14], 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 [4].

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, as well as the algorithm for inference of canonical structures, be presented in a formal, declarative fashion, which is currently not the case. To somewhat remedy the situation, and help the reader interested in developing their own overloaded lemmas, we include in our online appendix [11] a brief description of the order in which canonical instances are resolved in Coq. This is essentially a specification of the "operational semantics" of canonical instance resolution, and we have found it invaluable, but it has been obtained by a diligent study of the source code of Coq's unification algorithm.

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. In contrast, in Ltac, or in type classes, variable assignment is essentially constant-time.

Thus, even though the proof terms produced by application of our overloaded lemmas are usually much shorter than the proofs generated by corresponding tactics, they often take somewhat longer to generate and type check, and scale much worse. 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.

Acknowledgments

We would like to thank Cyril Cohen and Matthieu Sozeau for very helpful discussions.

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *TPHOLs*, volume 5674 of *LNCS*, pages 84–98, 2009.

[2] Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. CoqMTU: a higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *LICS*, pages 143–151, 2011.

[3] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101, 2008.

[4] Thomas Braibant and Damien Pous. Rewriting modulo associativity and commutativity in Coq. In *Second Coq workshop*, 2010.

[5] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP*, pages 241–253, 2005.

[6] Adam Chlipala. Certified programming with dependent types. URL: `http://adam.chlipala.net/cpdt`, 2008.

[7] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[8] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–93, 2008.

[9] Georges Gonthier. Point-free, set-free concrete linear algebra. In *ITP*, 2011.

[10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[11] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc, 2011. Code + appendix: `http://www.mpi-sws.org/~beta/lessadhoc`.

[12] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113, 2005.

[13] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *TOPLAS*, 18:241–256, 1996.

[14] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *POPL*, pages 275–286, 2010.

[15] Mark P. Jones. Type classes with functional dependencies. In *ESOP*, pages 230–244, 2000.

[16] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *CACM*, 53(6):107–115, 2010.

[17] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52:107–115, July 2009.

[18] J. Garrett Morris and Mark P. Jones. Instance chains: Type class programming without overlapping instances. In *ICFP*, pages 375–386, 2010.

[19] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274, 2010.

[20] Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *PPDP*, pages 163–173, 2008.

[21] Adam Poswolsky and Carsten Schürmann. System description: Delphin – a functional programming language for deductive systems. *ENTCS*, 228:113–120, 2009.

[22] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

[23] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *POPL*, pages 292–301, 1997.

[24] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293, 2008.

[25] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *MSCS, Special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21:1–31, 2011.

[26] Antonis Stampoulis and Zhong Shao. VeriML: Typed computation of logical terms inside a language with effects. In *ICFP*, pages 333–344, 2010.

[27] Pierre-Yves Strub. Coq modulo theory. In *CSL*, pages 529–543, 2010.

[28] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, pages 60–76, 1989.