Ynot : Reasoning with the Awkward Squad

Aleksandar Nanevski
Microsoft Research
aleksn@microsoft.com

Greg Morrisett Avi Shinnar
Paul Govereau
Harvard University
{greg, shinnar, govereau}@eecs.harvard.edu

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Abstract

We describe an axiomatic extension to the Coq proof assistant, that supports writing, reasoning about, and extracting higher-order, dependently-typed programs with *side-effects*. Coq already includes a powerful functional language that supports dependent types, but that language is limited to pure, total functions. The key contribution of our extension, which we call Ynot, is the added support for computations that may have effects such as non-termination, accessing a mutable store, and throwing/catching exceptions.

The axioms of Ynot form a small trusted computing base which has been formally justified in our previous work on Hoare Type Theory (HTT). We show how these axioms can be combined with the powerful type and abstraction mechanisms of Coq to build higher-level reasoning mechanisms which in turn can be used to build realistic, verified software components. To substantiate this claim, we describe here a representative series of modules that implement imperative finite maps, including support for a higher-order (effectful) iterator. The implementations range from simple (e.g., association lists) to complex (e.g., hash tables) but share a common interface which abstracts the implementation details and ensures that the modules properly implement the finite map abstraction.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Type Theory, Hoare Logic, Separation Logic, Monads

1. Introduction

Two main properties make type systems an effective and scalable formal method. First, important classes of programming errors are eliminated by statically enforcing the correct use of values. Second, types facilitate modular software development by serving as specifications of program components, while hiding the component's actual implementation. Implementations with the same type can be interchanged, improving software development, reuse and evolution.

Mainstream type systems focus on relatively simple properties that admit type inference and checking with little or no input from the programmer. Unfortunately, this leaves a number of properties, including data structure invariants, and API protocols, outside of their reach, and also restricts the practical programming features that can be safely supported. For example, most simply-typed languages cannot safely allow low-level operations such as pointer arithmetic or explicit memory management.

Static verification can be somewhat extended by refining types with annotations from more expressive logics, as implemented in ESC [25], JML [7], Spec# [2], Cyclone [20], Sage [14], DML [50], Deputy [8] and Liquid types [39]. However, because of the undecidability of the annotation logics, these systems still have trouble automatically discharging all but the most shallow specifications, and do not provide alternatives to the programmer when the automation fails, beyond run-time checks.

On the other hand, dependent type theories such as the Calculus of Inductive Constructions (implemented in the Coq proof assistant), can provide very strong correctness assurances, ranging from simple type safety to full functional correctness, and if some property cannot be discharged automatically, the programmer can provide the proof by hand.

Unfortunately, dependent type theories such as Coq, where proofs are represented as language terms, make it difficult to incorporate computational effects. For example, a diverging term, which can be assigned any type, could be presented as a "proof" of False, which renders the theory inconsistent. As a result, most type theories limit computations to total, pure functions with absolutely no side effects. For example, Coq excludes general recursion, mutable state, exceptions, I/O and concurrency. While some programming tasks, such as a compiler or a decision procedure, can be formulated as a pure, terminating function, most important tasks cannot. Furthermore, even computations that can be cast as pure functions need to use asymptotically efficient algorithms and mutable data structures (e.g., hash tables) to be practical.

In this paper, we present the system *Ynot*, which is our proposal for addressing the above shortcomings. Ynot starts with the type theory of Coq, which already has good support for functional programming, inductive definitions, specifications, proofs, and tactics, and extends it with a new type $\mathsf{ST}\,p\,A\,q$ and the associated programming constructors. Intuitively, ST classifies delayed, possibly effectful computations, much the way that the IO-monad classifies effectful computations in Haskell. The monadic separation facilitates a clean interaction between effects and pure Coq, preserving the logical soundness of the combined system. Unlike Haskell, our monad is indexed not only by the return type of the compu-

XXXX XXXX

tation, but also a precondition and postcondition, thus capturing Hoare Logic specifications in the type: When a computation of type $\mathsf{ST}\, p\, A\, q$ is run in a heap h_1 satisfying $p\, h_1$, and it terminates, it will produce a value x of type A and result in a new heap h_2 such that the predicate $q\, x\, h_1\, h_2$ holds. Computations can allocate, read, write, and deallocate locations, throw and catch exceptions, and perform general recursion. Thus, Ynot makes it possible to write ML or Haskell-style programs and yet still be able to formally reason about their values and effects.

Arranging Hoare-style specifications as monadic types leads to a very convenient formulation whereby the inference rules directly serve as monadic term constructors with specific operational behavior. In such a setting, program verification becomes directed by the syntax of the program, which has several important consequences. First, there is no need for annotation burden in the form of ghost state or auxiliary effectful code, which are often used in Hoare-style logics to expose intermediate program values and invariants and bring them into the scope of pre- and postconditions [36].

Second, a proof of any particular program component becomes independent of the context in which the program appears, and thus does not break when the program is considered in a larger context. In most other languages, this kind of *modular soundness* is non-trivial to achieve (if even possible), as argued for example by Leino and Nelson [24]. In particular, it implies that one can build libraries of *certified* code, thus reusing both the code and its correctness proof.

Third, the high-level structure of a proof for a stateful program becomes easier, since at each program point there is only one possible proof step to consider. Of course, applying the step may not be automatic, as we may need to supply witnesses for existential quantifiers. However, uniqueness eliminates the need for backtracking, and leads to a particularly compact way to arrange and *verify* the search process itself. In related verification systems such as ESC [10], WHY [13], Spec#[2] or Sage [14], this would correspond to verifying the verification condition generator – a step that, to the best of our knowledge, none of the above systems undertake.

Another important property that is inherent in Coq, and thus inherited by Ynot, is that types, propositions and proofs are first-class objects, which one can abstract, compute with, and compose into aggregate structures. The programmer can thus implement his or her own verified algorithms for automation of certain reasoning patterns [17] and scale the verification to programs of realistic size [26].

Many proposals for formal reasoning about stateful programs have been studied before, and even implemented in Coq. For example, Shao et al. [40, 34, 12], Benton and Zarfati [4] and Hobor et al. [18] consider simple imperative or low-level languages with a number of important features such as code pointers or concurrency, and formalize deep embeddings of Hoare and Separation Logics for such languages in Coq. Filliatre [13] presents a tool for generating verification conditions in the form of Coq propositions, but the programs supported by the tool do not admit higher-order abstractions or pointer aliasing. Boulme [6] shallowly embeds refinement theory into Coq, where one can reason about the refinement relation between two effectful programs, but does not seem to currently support adequate reasoning about pointer aliasing or locality of state. Kleymann [22] shows a shallow embedding of Hoare Logic in the related type theory of LEGO. Concoqtion by Fogarty et al. [15] is an effectful language whose types embed Coq propositions so as to reason precisely about program values, but cannot currently support reasoning about program effects. Marti and Affeldt in [28] present a deep embedding of Separation Logic in Coq, and certify the verification procedures for it.

Ynot differs from the above approaches in that it *axiomatically* extends Coq with selected monadic primitives. Of course, we have proved in the previous work [32, 31, 37] that these axioms do not cause unsoundness, and moreover, that the operational semantics of Ynot respects the ST specifications. Ynot differs from most of these systems in that it supports Hoare and Separation Logic reasoning for a higher-order, higher-typed language, where effectful computations are themselves first-class values that can e.g., be stored in references. Ynot differs from deep embeddings of Hoare Logic into Coq in that it can directly use Coq's purely functional fragment for programming. For example, a deep embedding of a language with higher-order functions must develop from scratch the type checking and reasoning rules about such functions. Ynot, on the other hand, immediately inherits all the purely-functional primitives and their reasoning principles such as extensionality, induction, primitive recursion, libraries of data types, lemmas, parametric polymorphism, existential types, quantification over type and predicate *constructors*, type equalities, etc. All of these, of course, are irreplaceable as tools for structuring, modularity and functorization of code and proofs.

Ynot also inherits Coq's built-in support for inference of types, annotations and implicit parameters. When combined with the monadic rules which we designed to avoid ghost state and auxiliary code, these lead to a programming style whose look-and-feel is very close to the current practice of languages like Haskell.

The abstractions of Ynot are not only directly useful in programming, but in defining *new* monads for new Hoare Logics. For example, we show how to define the monad STsep for specifications in the style of Separation Logic, directly in terms of ST, verifying in Coq all the steps along the way. Separation Logic is usually taken as a foundation for modular reasoning about state, but our ST monad shows that modularity can be achieved without starting from separation.

There will be no meta-theoretic proofs in the current paper; for that we refer the interested reader to our previous work [32, 31, 37]. The contribution of the current paper is to illustrate how the theory is translated into practice. In particular:

- We describe the design of Ynot including the primitive terms and their specifications, and show how they can be used to write effectful code in a Haskell style.

- We show how the design supports automatic inference of principal specifications for (loop-free) code, and eases the verification burden through the use of syntax-directed lemmas and tactics.

- We show how a separation monad, STsep can be defined on top of the ST primitives and used to build and modularly verify higher-order components.

- We describe a representative library of imperative, finite map implementations and their common interface. The library includes three implementations: association lists, hash-tables, and splay-trees. The libraries support a higher-order operation for iterating an effectful computation over the map, and reasoning about the compound effects of the iterator. We also describe a verified memoization library that we have constructed, based on the finite map interface.

- We describe our prototype compiler that maps Ynot code to the Glasgow Haskell Compiler's intermediate language. The compiler has been used to compile all of the sample code in our library.

The source code of our Coq development can be found at `http://www.eecs.harvard.edu/~greg/ynot`. To clarify the presentation and fit into the allotted space, the code presented here is close, but not exactly the same as the library. In particular, we have formatted the code using suggestive mathematical notation, and omitted the proof scripts.

2. Basics of Ynot

We begin by showing some simple imperative Ynot code. The definitions below implement an imperative finite map using a reference to an (immutable) association list. They take advantage of previously-defined key and list operations such as cmp, remove and assoc which compare keys, remove key-value pairs from a list, and look up the value associated with a key respectively.

```
Definition create := new empty_kv_list.

Definition insert r k v :=
  ℓ ← read r; write r ((k, v) :: (remove cmp k ℓ)).

Definition delete r k :=
  ℓ ← read r; write r (remove cmp k ℓ).

Definition lookup r k :=
  ℓ ← read r;
  match_option (assoc cmp k ℓ)
    (throw Failure)
    (λv ⇒ ret v).

Definition destroy r := free r.
```

The code is written in a style similar to Haskell's "do" notation. In particular, the notation "$x \leftarrow e_1; e_2$" is short-hand (definable in Coq) for "bind e_1 $(\lambda x.e_2)$" and is used to construct sequential compositions of primitive computations. In this case, the primitive computations include effectful operations such as new (for allocating and initializing a reference), read and write (for accessing the contents of a reference), free (for deallocating a reference), and throw for throwing an exception. The other key primitives include ret, which lifts a pure Coq value into the space of computations; fix, which is used to construct recursive computations; and try, which is used to catch exceptions.

Also like Haskell, Ynot can automatically infer types for the code. The types of effectful computations will in general be classified by a monadic type construct ST. However, unlike Haskell, the ST monad is indexed by pre- and postconditions that summarize when it is safe to run a computation, and what the effects of the computation are on the world. For example, it is not safe to run the read computation on a ref x that has been free'd. In reality, this inference only works for straightline code and so the specifications of loops using fix must be written explicitly. Nevertheless, the inference mechanisms make it relatively easy to write useful imperative code, abstract over it, and reason about it.

In what follows, we describe more formally the basics of the Ynot primitives, pre- and postconditions, our model of heaps, etc. We then show how programmers can write code with explicit specifications and convince the type-checker, through explicit proofs, that code respects a given interface. The rest of the paper shows how, using these facilities and the abstraction mechanisms of Coq, we can realize an abstract interface for finite maps that supports not only this simple association list, but also more sophisticated data structures, including hash tables and splay trees. We also consider higher-order operations, such as a fold over the finite map, that supports effectful computations.

2.1 Formalism

The design of Ynot is a generalization of the well-known ideas from type-and-effect systems [16] and their monadic counterparts [30, 44]. Our monadic type ST $p\,A\,q$ classifies programs that return values of type A, but that may also perform stateful side effects. The effect annotations p and q take the role of a precondition and a postcondition. They are drawn from higher-order logic, rather than from any particular finitary algebraic structure, as usually the case in simple type-and-effects. The expressiveness of higher-order logic will allow us to very precisely track the stateful behavior of programs. Of course, because p and q may *depend* on run-time entities like memory locations and exceptions, this precise tracking essentially requires dependent types.

More formally, the signature of the ST type constructor is as follows. To reduce clutter, here we use a more stylized mathematical notation instead of the actual concrete syntax of Coq.

$$\mathsf{ST} : \mathsf{pre} \rightarrow \Pi A{:}\mathsf{Type}.\ \mathsf{post}\ A \rightarrow \mathsf{Type}$$

Here pre is the type of preconditions, and post A is the type of postconditions, where the postconditions also scopes over the answer that the computation returns.

$$\mathsf{pre} := \mathsf{heap} \rightarrow \mathsf{Prop} \qquad \mathsf{post}\ A := \mathsf{ans}\ A \rightarrow \mathsf{heap} \rightarrow \mathsf{heap} \rightarrow \mathsf{Prop}$$

The answer of a computation can either be a value (of some type A) or an exception that the computation raised. We distinguish between these two cases using the type ans A and its two constructors: $\mathsf{Val} : A \rightarrow \mathsf{ans}\ A$ and $\mathsf{Exn} : \mathsf{exn} \rightarrow \mathsf{ans}\ A$.

Heaps are modeled as partial functions from locations to type dynamic. Elements of dynamic are records, packaging a type and a value, thus abstracting the type. Modeling heaps this way allows us to implement strong updates, whereby a location can point to values of varying types.

```
dynamic := {type:Type, val:type}
heap := loc → option dynamic
empty : heap := λℓ. None
update_loc (h:heap)(r:loc)(v:A) : heap :=
    λx:loc. if r == x then Some {type=A, val=v} else h x
```

The types loc and exn denote locations and exceptions, respectively. For each type, we assume they are countably infinite and support decidable equality. In a model, they can be treated as isomorphic to natural numbers.

As customary (e.g, in Separation Logic), preconditions are unary relations over heaps, specifying the set of heaps in which the program can execute without causing any memory errors such as dereferencing a dangling pointer. Postconditions relate the answer of a computation with *both* the initial and ending heaps. Hoare Logic postconditions usually range only over the ending state, but formulations that range over both states are not uncommon, and date back at least to the classical work on the Vienna Development Method [21]. It is well-known that the latter obviates the need for *ghost* variables, which scope over precondition *and* postcondition and serve to relate the old with the new state. Ghost variables are unwieldy in the presence of higher-order abstraction, because they either have too large a scope (usually global, thus interfering with modularity) or else the user has to track their scope and provide explicit instantiations. That is why we avoid them by adopting binary postconditions. As an example, the code "$n \leftarrow$ read x; write x $(n+1)$; ret n" which increments a location x and then returns the original value in x, can be given a postcondition like the following:

$$\lambda(a{:}\mathsf{ans\ nat})\,(i\ f{:}\mathsf{heap}).\,\forall n.\,i\,x = \mathsf{Some}\{\mathsf{type}{=}\mathsf{nat}, \mathsf{val}{=}n\} \rightarrow a = \mathsf{Val}\,n \wedge f = \mathsf{update\_loc}\,i\,x\,(n+1)$$

The postcondition says that if in the initial memory i the location x maps to some natural number n, then the answer returned by the computation will be the value n, and the final memory will be equivalent to updating the initial memory at location x with the value $n+1$. Note that we need the initial state i to correctly specify the return value as well as the final state.

The constructors of the ST monad and their signatures are given in Figure 1. As we pointed out previously, these are added axiomatically to Coq, but we have proved in [32, 37] that it is sound to do so. These will be the only axioms of Ynot, and the rest of the development presented in this paper is completely carried out definitionally within Coq.

In Figure 1 and further in the text we take the customary notational liberties. For example, we use infix notation for $x \hookrightarrow_A v$

$$\mathsf{ret} : \Pi x{:}A.\ \mathsf{ST}\ \mathsf{top}\ A\ (\lambda a\, i\, f.\ f = i \wedge a = \mathsf{Val}\, x)$$

$$\begin{array}{l}\mathsf{bind} : \mathsf{ST}\ p_1\ A\ q_1 \to (\Pi x{:}A.\ \mathsf{ST}\ (p_2\, x)\ B\ (q_2\, x)) \to \\ \quad \mathsf{ST}\ (\lambda i.\ p_1\, i \wedge \forall x\, h.\ q_1\, (\mathsf{Val}\, x)\, i\, h \to p_2\, x\, h)\ B \\ \qquad (\lambda a\, i\, f.\ ((\exists x\, h.\ q_1\, (\mathsf{Val}\, x)\, i\, h \wedge q_2\, x\, a\, h\, f) \vee \\ \qquad\qquad \exists e.\ a = \mathsf{Exn}\, e \wedge q_1\, (\mathsf{Exn}\, e)\, i\, f)\end{array}$$

$$\begin{array}{l}\mathsf{do} : \mathsf{ST}\ p_1\ A\ q_1 \to ((\forall i.\ p_2\, i \to p_1\, i) \wedge \\ \qquad\qquad \forall a\, i\, f.\ p_2\, i \to q_1\, a\, i\, f \to q_2\, a\, i\, f) \to \mathsf{ST}\ p_2\ A\ q_2\end{array}$$

$$\begin{array}{l}\mathsf{read} : \Pi r{:}\mathsf{loc}.\ \mathsf{ST}\ (r \hookrightarrow_A -)\ A\ (\lambda a\, i\, f.\ f = i \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall v{:}A.\ (r \hookrightarrow v)\, i \to a = \mathsf{Val}\, v)\end{array}$$

$$\begin{array}{l}\mathsf{write} : \Pi r{:}\mathsf{loc}.\ \Pi v{:}A.\ \mathsf{ST}\ (r \hookrightarrow -)\ \mathsf{unit}\ (\lambda a\, i\, f.\ a = \mathsf{Val}\, \mathsf{tt} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad f = \mathsf{update\_loc}\, i\, r\, v)\end{array}$$

$$\begin{array}{l}\mathsf{new} : \Pi x{:}A.\ \mathsf{ST}\ \mathsf{top}\ \mathsf{loc}\ (\lambda a\, i\, f.\ \exists r{:}\mathsf{loc}.\ a = \mathsf{Val}\, r \wedge i\, r = \mathsf{None} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad f = \mathsf{update\_loc}\, i\, r\, v)\end{array}$$

$$\mathsf{free} : \Pi r{:}\mathsf{loc}.\ \mathsf{ST}\ (r \hookrightarrow -)\ \mathsf{unit}\ (\lambda a\, i\, f.\ a = \mathsf{Val}\, \mathsf{tt} \wedge f = \mathsf{free\_loc}\, i\, r)$$

$$\mathsf{throw} : \Pi x{:}\mathsf{Exn}.\ \mathsf{ST}\ \mathsf{top}\ A\ (\lambda a\, i\, f.\ f = i \wedge a = \mathsf{Exn}\, x)$$

$$\begin{array}{l}\mathsf{try} : \mathsf{ST}\ p_1\ A\ q_1 \to (\Pi x{:}A.\ \mathsf{ST}\ (p_2\, x)\ B\ (q_2\, x)) \to \\ \qquad (\Pi e{:}\mathsf{exn}.\ \mathsf{ST}\ (p_3\, e)\ B\ (q_3\, e)) \to \\ \quad \mathsf{ST}\ (\lambda i.\ p_1\, i \wedge (\forall x\, h.\ q_1\, (\mathsf{Val}\, x)\, i\, h \to p_2\, x\, h) \wedge \\ \qquad\qquad \forall e\, h.\ q_1\, (\mathsf{Exn}\, e)\, i\, h \to p_3\, e\, h)\ B \\ \qquad (\lambda a\, i\, f.\ (\exists x\, h.\ q_1\, (\mathsf{Val}\, x)\, i\, h \wedge q_2\, x\, a\, h\, f) \vee \\ \qquad\qquad \exists e\, h.\ q_1\, (\mathsf{Exn}\, e)\, i\, h \wedge q_3\, e\, a\, h\, f)\end{array}$$

$$\begin{array}{l}\mathsf{fix} : (\Pi x{:}A.\ \mathsf{ST}\ (p\, x)\ (B\, x)\ (q\, x) \to \Pi x{:}A.\ \mathsf{ST}\ (p\, x)\ (B\, x)\ (q\, x)) \\ \qquad \to \Pi x{:}A.\ \mathsf{ST}\ (p\, x)\ (B\, x)\ (q\, x)\end{array}$$

$$\begin{array}{l}\mathsf{with}\ \ \mathsf{top} := \lambda h.\ \mathsf{True} \\ \qquad x \hookrightarrow_A v := \lambda h{:}\mathsf{heap}.\ h\, x = \mathsf{Some}\, \{\mathsf{type}{=}A, \mathsf{val}{=}v\} \\ \qquad \mathsf{free\_loc}\, (h{:}\mathsf{heap})(r{:}\mathsf{loc}) : \mathsf{heap} := \lambda x{:}\mathsf{loc}.\ \mathsf{if}\ r == x\ \mathsf{then}\ \mathsf{None}\ \mathsf{else}\ h\, x\end{array}$$

Figure 1. Signature of ST primitives.

which is a predicate over heaps that holds of the heap h iff h contains a location x pointing to $v{:}A$. We also abbreviate its existential abstraction over v with $x \hookrightarrow_A -$, use $x \hookrightarrow -$ when abstracting over both v and A, and simply $x \hookrightarrow v$ when A can be inferred from the context. Similar abbreviations can be made in Coq as well, using definitions, implicit parameters, and notation declarations [29]. Thus, the example programs that we write in the rest of the paper in our stylized notation and with abbreviations, remain very close to the Coq implementation.

We next discuss the types from Figure 1. In English, ret is the monadic unit (Haskell's return). It takes a value $x{:}A$ and produces a computation that immediately returns x. The precondition top allows this computation to be executed in any heap. The postcondition $\lambda a\, i\, f.\ f = i \wedge a = \mathsf{Val}\, x$ guarantees that the final heap f equals the initial heap i and that the answer a equals x. The answer tag Val differentiates values from exceptions, so the postcondition also specifies that ret does not raise an exception. An interesting point to note here is that the range type of ret may be viewed as a somewhat stylized form of a Hoare rule for assignment of x to the variable a.

As in Haskell, bind is a construct for sequential composition. Given $e_1{:}\mathsf{ST}\, p_1\, A\, q_1$ and $e_2{:}\Pi x{:}A.\ \mathsf{ST}\ (p_2\, x)\ B\ (q_2\, x)$, $\mathsf{bind}\, e_1\, e_2$ first executes e_1 to obtain the result $x{:}A$, and then executes $e_2\, x$. Thus, it should not be surprising that the Ynot type for bind is very closely related to the Hoare rule for sequential composition. However, there are significant differences as well, which we introduced to facilitate inference. For example, the precondition of the composition $\lambda i.\ p_1\, i \wedge \forall x\, h.\ q_1\, (\mathsf{Val}\, x)\, i\, h \ \to\ p_2\, x\, h$, has a two-fold purpose. First, it requires that $(p_1\, i)$ holds so that e_1 can execute against the initial heap. Then, it requires that any return value $x{:}A$ and any $h{:}\mathsf{heap}$ that may have been obtained by the execution of e_1 – and which thus necessarily satisfy $q_1\, (\mathsf{Val}\, x)\, i\, h$ – must imply $(p_2\, x\, h)$, so that $e_2\, x$ can subsequently execute against the heap h. In most Hoare-style logics, the second conjunct is not part of the precondition, but is made unnecessary by requiring that q_1 and p_2 are syntactically equal. We decided against this formulation because, in practice, it forces the programmer into a potentially burdensome task of converting q_1 and p_2 into some syntactically equal form before the sequential composition could even be formed. Nevertheless, if this more traditional formulation is desired, it can easily be derived from our rule.

This derivability is guaranteed because our rule is, in a sense, the most primitive one, as it *computes a principal specification* for the sequential composition out of the specifications of the two components. Indeed, our precondition is essentially the weakest heap predicate that guarantees that the composition is safe. The computed postcondition is also principal, under the assumption that the whole composition terminates (ST captures only partial correctness semantics). The postcondition consists of two disjuncts that essentially describe the behavior of the composition if e_1 terminates with a value, and if e_1 terminates by raising an exception. In the first case, there exist a value $x{:}A$ and an intermediate heap h, obtained after the execution of e_1, which by typing of e_1 must satisfy $q_1\, (\mathsf{Val}\, x)\, i\, h$. If e_2 terminates when executed in h, then there also exists some final answer a – either a value or an exception – and some final heap f which, by the typing of e_2, must satisfy $q_2\, x\, a\, h\, f$. If e_1 raises an exception e, then the execution of e_2 is not even attempted. The exception is returned as the answer of the whole composition (conjunct $a = \mathsf{Exn}\, e$), and the final heap f is the one obtained after executing e_1 (conjunct $q_1\, (\mathsf{Exn}\, e)\, i\, f$).

The command $\mathsf{try}\, e\, e_1\, e_2$ is an exception-handling construct which executes e_2 if e raises an exception, but instead of falling through in case e returns a value, try actually proceeds with executing e_1. The utility of this monadic form of try has been argued by Benton and Kennedy [3] in compiler optimizations, where it facilitates code motion. We adopt this form for similar reasons, as rearranging code in semantic-preserving way can sometimes make proofs about programs much easier.

Example 1. Consider the delete operation given earlier (without the syntactic sugar, and with explicit declaration of argument types):

$$\begin{array}{l}\textbf{Definition}\ \mathsf{delete}\ (r{:}\mathsf{loc})\ (k{:}\mathsf{key})\ := \\ \qquad \mathsf{bind}\ (\mathsf{read}\ r)(\lambda \ell.\mathsf{write}\ r\ (\mathsf{remove\ cmp}\ \ k\ \ell)).\end{array}$$

and assuming that types key, value, kvlist := list(key∗value), have already been defined. The inferred type for delete is rather involved and is given by:

$$\begin{array}{l}\Pi r{:}\mathsf{loc}.\ \Pi k{:}\mathsf{key}. \\ \quad \mathsf{ST}\ (\lambda i.\ (\exists v{:}\mathsf{kvlist}.\ (r \hookrightarrow v)\, i) \wedge \forall x\, h.\ h = i \wedge \\ \qquad\qquad (\forall v{:}\mathsf{kvlist}.\ (r \hookrightarrow v)\, i \to \mathsf{Val}\, x = \mathsf{Val}\, v) \to (r \hookrightarrow -)\, h) \\ \qquad \mathsf{unit} \\ \qquad (\lambda a\, i\, f.\ (\exists v{:}\mathsf{kvlist}.\ (r \hookrightarrow v)\, i) \wedge \\ \qquad\quad ((\exists x\, h.\ (h = i \wedge (\forall v{:}\mathsf{kvlist}.\ (r \hookrightarrow v)\, i \to \mathsf{Val}\, x = \mathsf{Val}\, v)) \wedge \\ \qquad\qquad\quad a = \mathsf{Val}\, \mathsf{tt} \wedge f = \mathsf{update\_loc}\, h\, r\, (\mathsf{remove\ cmp}\, k\, x)) \vee \\ \qquad\qquad (\exists e.\ a = \mathsf{Exn}\, e \wedge f = i \wedge \\ \qquad\qquad\quad \forall v{:}\mathsf{kvlist}.\ (r \hookrightarrow v)\, i \to \mathsf{Exn}\, e = \mathsf{Val}\, v)))\end{array}$$

Of course, this is only one of many possible types, because there are many ways to syntactically represent semantically equivalent pre- and postconditions. Furthermore, a programmer may wish to not only use a syntactically different specification, but one where the precondition is stronger and/or the postcondition is weaker than the principle specification we infer. This is the role of the the do primitive, which corresponds to the rule of consequence in Hoare Logic. It takes a computation of type $\mathsf{ST}\, p_1\, A\, q_1$, and changes the type into $\mathsf{ST}\, p_2\, A\, q_2$, *if a proof is explicitly provided as an argument of do* that p_1 can be strengthened into p_2 and q_1 can be weakened into q_2. The idea is that p_2 and q_2 may provide a more abstract specification than p_1 and q_1, and hide unwanted implementation details. Hence, do is essential to information hiding.

Example 2. It is possible to use Coq's built-in "match" construct to build conditional monadic expressions. However, then we are forced to ensure that each branch of the conditional has the same type (up to definitional equivalence) when usually, the pre- and postconditions of the two branches differ. The user can avoid this problem by coding her own match primitive which uses "do" to coerce the branches to an ST type with common pre- and post-conditions. For example, the function match_option, used in the definition of our association list lookup, can be coded as follows:

```
Program Definition
match_option (x : option A) (c1 : ST p1 B q1)
             (c2 : Πv:A. ST (p2 v) B (q2 v)) :
    ST (λi. (x = None → p1 i) ∧ ∀v. x = Some v → p2 v i) B
       (λa i f. (x = None → q1 a i f) ∧
                ∀v. x = Some v → q2 v a i f) :=
match x with
| None ⇒ do c1 _
| Some v ⇒ do (c2 v) _
end.
```

Notice that the compound command has the pre- and postcondition corresponding to the Hoare-rule for conditionals, but that it is necessary to coerce both commands to the common specification so that Coq can type-check the code. Of course, once match_option is defined, we no longer have to worry about finding the weakest common precondition and strongest common postcondition.

The definition given here uses under-scores for the proof obligations of the two uses of do. The recent Russel extensions to Coq allow us to omit proof obligations and fill them in after constructing a definition [42]. In this case, once we have entered the definition of match_option, we are left with two obligations, which must show that the two commands' specifications can be weakened to the common interface, given the outcome of the test. The two obligations are discharged with explicit proof scripts as follows:

```
Next Obligation. firstorder; discriminate. Qed.
Next Obligation.
  split; try solve[firstorder].
  intros x i m [H1 H2] H; split; try solve[intros; discriminate].
  intros v0 t; injection t; intros []; auto.
Qed.
```

Example 3. As a final example in this section, and one that is far more typical in the code we have written, we can use do to assign our delete operation an explicit type with a much more readable (though in this case weaker) specification than the principal one inferred by Ynot:

```
Program Definition delete (r:loc) (k:key) :
    ST (r ↪kvlist −) unit
       (λa i f. a = Val tt ∧
                ∀v:kvlist. (r ↪ v) i → (r ↪ remove cmp k v) f)
:= do (ℓ ← read r; write r (remove cmp k ℓ)) _.

Next Obligation.
  ...
Qed.
```

2.2 A Monad for Separation Logic

The weakened specification of delete from Example 3 is unfortunately too weak. To see this, consider the scenario in which we want to remove keys from two separate association lists, e.g.:

```
delete_two(r1 r2:loc) (k1 k2:key) := delete r1 k1; delete r2 k2
```

Assume further that r_1 and r_2 contain association lists and are not aliased in the initial heap, that is $(r_1 \hookrightarrow_{\mathsf{kvlist}} -)\, i$, $(r_2 \hookrightarrow_{\mathsf{kvlist}} -)\, i$ and $r_1 \neq r_2$. One may intuitively think that these assumptions suffice to prove that delete_two is safe, but that is not the case. To verify delete_two, Ynot will first combine the inference rule for bind from Figure 1 with the specification of delete to infer the precondition:

$$P = \lambda i.\, (r_1 \hookrightarrow -)\, i \wedge \forall x\, h.\, (x = \mathsf{Val}\,\mathsf{tt} \wedge \forall v.\, (r_1 \hookrightarrow v)\, i \rightarrow (r_1 \hookrightarrow \mathsf{remove}\,\mathsf{cmp}\, k_1\, v)\, h) \rightarrow (r_2 \hookrightarrow -)\, h$$

Then it will require a proof that $P\, i$ is valid under the described assumptions, which amounts to inferring the conclusion $(r_2 \hookrightarrow -)\, h$, from $(r_2 \hookrightarrow -)\, i$ and the constraint that $(r_1 \hookrightarrow v)\, i \rightarrow (r_1 \hookrightarrow \mathsf{remove}\,\mathsf{cmp}\, k_1\, v)\, h$. Such inference cannot be made, of course, as the constraint does not relate h and r_2 in any way.

The problem, as already identified by the work on Separation Logic [35] is that the weakened postcondition of delete describes how location r is changed, but forgets to specify that the the rest of the heap, disjoint from location r, remains intact.

A stronger and appropriate specification for delete can certainly be written directly in the ST monad. However, the illustrated properties of disjointness and heap invariance appear so often in practice that it is better to introduce a general and uniform methodology for dealing with them.

We therefore introduce a new monad STsep to codify the reasoning in Separation Logic. Informally, the idea is that e : STsep $p\, A\, q$ should hold iff whenever e is executed in a heap $i \uplus h$, where i and h are disjoint and $p\, i$ holds, then if e terminates, we are left in a state $f \uplus h$ such that $q\, i\, f$ holds. That is, the h portion of the state, which is not covered by the pre-condition, remains unaffected by default. In Separation Logic, programs like this, which do not touch the heap outside of what their precondition circumscribes, are said to satisfy the *frame property*.

Fortunately, the STsep monad can be defined in terms of ST and does *not* need to be added axiomatically.

$$\mathsf{STsep} : \mathsf{pre} \rightarrow \Pi A{:}\mathsf{Type}.\ \mathsf{post}\, A \rightarrow \mathsf{Type}$$

$$\mathsf{STsep}\, p\, A\, q = \mathsf{ST}\, (p * \mathsf{top})\, A\, (p \multimap q)$$

Here $*$ is the well-known separating conjunction, and $p \multimap q$ builds a postcondition which essentially states that q only describes how to change the parts of the initial heap circumscribed by p.

$$P_1 * P_2 = \lambda h{:}\mathsf{heap}.\, \exists h_1, h_2.\, \mathsf{splits}\, h\, h_1\, h_2 \wedge P_1\, h_1 \wedge P_2\, h_2$$
$$P \multimap Q = \lambda i\, m.\, \forall i_1\, h.\, P\, i_1 \rightarrow \mathsf{splits}\, i\, i_1\, h \rightarrow \exists m_1.\, \mathsf{splits}\, m\, m_1\, h \wedge Q\, i_1\, m_1$$

where

```
splits (h h1 h2:heap) := disjoint h1 h2 ∧ h = union h1 h2
union (h1 h2:heap) : heap :=
    λx:loc. match h1 x with None ⇒ h2 x | _ ⇒ h1 x end
disjoint (h1 h2:heap) := ∀x:loc. (x ↪ −) h1 → h2 x = None
```

The definition of STsep therefore directly formalizes the intuition behind separation logic and the frame property. In this sense, it is closely related to the recent semantic models of Separation Logic [5] in which the frame rule is "baked in" to the interpretation of triples. The fact that STsep could be defined in terms of ST may be surprising, because it shows that even in the large footprint semantics of ST, Ynot programs do satisfy the frame property. The inference rules of ST compute the weakest precondition that guarantees that the program is safe, meaning that the inferred precondition must basically represent the memory footprint of the program. The only way in which the precondition can be changed, is by do-encapsulation and this can only strengthen the precondition. Thus, in both of our monads, a memory operation cannot be performed unless the access to the particular location is, intuitively, "granted by the precondition". This is, of course, the essence of the modular nature of Separation Logic and of Ynot.

The next step in the definition is to re-type each of the ST primitives, so that they can be used in the STsep monad. This can be done in each case by an appeal to the rule of consequence, as shown in our Coq implementation. Here, we only give the new signatures in Figure 2.

Separation Logic requires a separate rule – the *frame rule* – to make explicit logical inferences with the frame property. In the

$$\mathsf{ret} : \Pi x{:}A.\ \mathsf{STsep}\ \mathsf{emp}\ A\ (\lambda a\ i\ f.\ f = i \wedge a = \mathsf{Val}\ x)$$

$$\begin{array}{l}\mathsf{bind} : \mathsf{STsep}\ p_1\ A\ q_1 \to (\Pi x{:}A.\ \mathsf{STsep}\ (p_2\ x)\ B\ (q_2\ x)) \to \\ \quad \mathsf{STsep}\ (\lambda i.\ (p_1 * \mathsf{top})\ i \wedge \forall x\ h.\ (p_1 \multimap q_1)\ (\mathsf{Val}\ x)\ i\ h \to \\ \qquad\qquad (p_2 * \mathsf{top})\ x\ h)\ B \\ \qquad (\lambda a\ i\ f.\ (\ ((\exists x\ h.\ (p_1 \multimap q_1)\ (\mathsf{Val}\ x)\ i\ h \wedge (p_2 \multimap q_2)\ x\ a\ h\ f) \vee \\ \qquad\qquad \exists e.\ a = \mathsf{Exn}\ e \wedge (p_1 \multimap q_1)\ (\mathsf{Exn}\ e)\ i\ f)\end{array}$$

$$\begin{array}{l}\mathsf{do} : \mathsf{STsep}\ p_1\ A\ q_1 \to (\forall i.\ p_2\ i \to \exists h.\ (p_1 *\ \mathsf{this}\ h)\ i \wedge \\ \qquad \forall a\ f.\ (q_1\ a * \mathsf{delta}(\mathsf{this}\ h))\ i\ f \to q_2\ a\ i\ f) \to \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{STsep}\ p_2\ A\ q_2\end{array}$$

$$\begin{array}{l}\mathsf{read} : \Pi r{:}\mathsf{loc}.\ \mathsf{STsep}\ (r \mapsto_A -)\ A\ (\lambda a\ i\ f.\ f = i \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall v{:}A.\ (r \mapsto v)\ i \to a = \mathsf{Val}\ v)\end{array}$$

$$\mathsf{write} : \Pi r{:}\mathsf{loc}.\ \Pi v{:}A.\ \mathsf{STsep}\ (r \mapsto -)\ \mathsf{unit}\ (\lambda a\ i\ f.\ (r \mapsto v)\ f \wedge a = \mathsf{Val}\ \mathsf{tt})$$

$$\mathsf{new} : \Pi x{:}A.\ \mathsf{STsep}\ \mathsf{emp}\ \mathsf{loc}\ (\lambda a\ i\ f.\ \exists r{:}\mathsf{loc}.\ a = \mathsf{Val}\ r \wedge (r \mapsto x)\ f)$$

$$\mathsf{free} : \Pi r{:}\mathsf{loc}.\ \mathsf{STsep}\ (r \mapsto -)\ \mathsf{unit}\ (\lambda a\ i\ f.\ a = \mathsf{Val}\ \mathsf{tt} \wedge \mathsf{emp}\ f)$$

$$\mathsf{throw} : \Pi x{:}\mathsf{Exn}.\ \mathsf{STsep}\ \mathsf{emp}\ A\ (\lambda a\ i\ f.\ f = i \wedge a = \mathsf{Exn}\ x)$$

$$\begin{array}{l}\mathsf{try} : \mathsf{STsep}\ p_1\ A\ q_1 \to (\Pi x{:}A.\ \mathsf{STsep}\ (p_2\ x)\ B\ (q_2\ x)) \to \\ \qquad (\Pi e{:}\mathsf{exn}.\ \mathsf{STsep}\ (p_3\ e)\ B\ (q_3\ e)) \to \\ \quad \mathsf{STsep}\ (\lambda i.\ (p_1 * \mathsf{top})\ i \wedge \\ \qquad\qquad (\forall x\ h.\ (p_1 \multimap q_1)\ (\mathsf{Val}\ x)\ i\ h \to (p_2 * \mathsf{top})\ x\ h) \wedge \\ \qquad\qquad\ \forall e\ h.\ (p_1 \multimap q_1)\ (\mathsf{Exn}\ e)\ i\ h \to (p_3 * \mathsf{top})\ e\ h)\ B \\ \qquad\qquad (\lambda a\ i\ f.\ (\exists x\ h.\ (p_1 \multimap q_1)\ (\mathsf{Val}\ x)\ i\ h \wedge (p_2 \multimap q_2)\ x\ a\ h\ f) \vee \\ \qquad\qquad\ \exists e\ h.\ (p_1 \multimap q_1)\ (\mathsf{Exn}\ e)\ i\ h \wedge (p_3 \multimap q_3)\ e\ a\ h\ f)\end{array}$$

$$\begin{array}{l}\mathsf{fix} : (\Pi x{:}A.\ \mathsf{STsep}\ (p\ x)\ (B\ x)\ (q\ x) \to \Pi x{:}A.\ \mathsf{STsep}\ (p\ x)\ (B\ x)\ (q\ x)) \\ \qquad \to \Pi x{:}A.\ \mathsf{STsep}\ (p\ x)\ (B\ x)\ (q\ x)\end{array}$$

$$\begin{array}{lrll}\mathsf{where} & \mathsf{emp} & := & \lambda h.\ h = \mathsf{empty} \\ & x \mapsto_A v & := & \lambda h.\ h = \mathsf{update\_loc}\ \mathsf{empty}\ x\ v \\ & \mathsf{this} & := & \lambda h_1\ h_2.\ h_1 = h_2 \\ & \mathsf{delta}\ P & := & \lambda h_1\ h_2.\ P\ h_1 \wedge P\ h_2 \\ & Q_1 * Q_2 & := & \lambda i\ h.\ \exists i_1\ i_2\ h_1\ h_2.\ \mathsf{splits}\ i\ i_1\ i_2 \wedge \\ & & & \qquad \mathsf{splits}\ h\ h_1\ h_2 \wedge Q_1\ i_1\ h_1 \wedge Q_2\ i_2\ h_2\end{array}$$

Figure 2. Signature of STsep primitives.

STsep monad, that role is ascribed to the constructor do, whose type now encodes not only the rule of consequence, but the frame rule as well. For example, in order to weaken $\mathsf{STsep}\ p_1\ A\ q_1$ into $\mathsf{STsep}\ p_2\ A\ q_2$, we now require showing that there exists a subheap h which remains invariant across the computation. If h is restricted to be empty, then we recover the old ST rule of consequence.

The important point, however, is that switching from ST to STsep does not require adding extra inference rules and constructs, and the programs in STsep retain the same general shape and operational behavior as their ST counterparts (though the specifications and correctness proofs may change).

Example 4. The proper type for delete in the STsep monad is

$$\begin{array}{l}\mathsf{delete}\ (r{:}\mathsf{loc})\ (k{:}\mathsf{key}) : \\ \quad \mathsf{STsep}\ (r \mapsto_{\mathsf{kvlist}} -)\ \mathsf{unit} \\ \qquad (\lambda a\ i\ f.\ a = \mathsf{Val}\ \mathsf{tt} \wedge \\ \qquad\quad \forall v{:}\mathsf{kvlist}.\ (r \mapsto v)\ i \to (r \mapsto \mathsf{remove\ cmp}\ k\ v)\ f)\end{array}$$

This differs from Example 3 only in that $\hookrightarrow$ is replaced by the more precise points-to relation $\mapsto$. That suffices to type check delete_two as well.

$$\begin{array}{l}\textbf{Program Definition}\ \mathsf{delete\_two}\ (r_1\ r_2{:}\mathsf{loc})\ (k_1\ k_2{:}\mathsf{key}) : \\ \quad \mathsf{STsep}\ (r_1 \mapsto_{\mathsf{kvlist}} - * r_2 \mapsto_{\mathsf{kvlist}} -)\ \mathsf{unit} \\ \qquad (\lambda a\ i\ f.\ a = \mathsf{Val}\ \mathsf{tt} \wedge \\ \qquad\quad \forall v_1\ v_2.\ ((r_1 \mapsto v_1) * (r_2 \mapsto v_2))\ i \to \\ \qquad\quad ((r_1 \mapsto \mathsf{remove\ cmp}\ k_1\ v_1) * (r_2 \mapsto \mathsf{remove\ cmp}\ k_2\ v_2))\ f) \\ := \ \mathsf{do}\ (\mathsf{delete}\ k_1\ v_1; \mathsf{delete}\ k_2\ v_2)\ \_.\end{array}$$

2.3 Verification

The inference rules from Figures 1 and 2 may look rather unwieldy at first sight, especially when compared to rules in Hoare or Separation Logic. This is just as well; we do not intend to use the rules manually in verification, but to infer the principle specification of monadic programs (up to loop invariants). In this sense, Ynot is really more closely related to predicate transformers [11] than to Hoare Logic. The annotation inference essentially *compiles* the program into a Coq proposition whose validity implies the program's correctness, thereby reducing verification to theorem proving in Coq. This principal proposition may be rather large and difficult to tackle directly. A useful simplifying strategy is to divide it into a set of smaller verification conditions, which are easier to prove.

In this process, Ynot exploits the property that the principal proposition inherits the structure of the program it is generated from. Consider a program of the form $\mathsf{bind}\ (\mathsf{read}\ x)\ e$, where $e : \Pi y{:}A.\ \mathsf{STsep}\ (p_2\ y)\ B\ (q_2\ y)$ is arbitrary, and suppose we want to coerce this program into the type $\mathsf{STsep}\ p_1\ B\ q_1$. The inference rules from Figure 2 would require a proof obligation of the form

$$\begin{array}{l}\forall i{:}\mathsf{heap}.\ p_1\ i \to \\ \quad \mathsf{verifies}\ i\ (\mathsf{bind\_pre}\ (\mathsf{read\_pre}\ A\ x)\ (\mathsf{read\_post}\ A\ x)\ p_2) \\ \qquad\qquad (\mathsf{bind\_post}\ (\mathsf{read\_pre}\ A\ x)\ (\mathsf{read\_post}\ A\ x)\ p_2\ q_2) \\ \qquad\qquad q_1\end{array}$$

Here verifies abbreviates part of the obligation in the STsep typing rule for do; that is:

$$\begin{array}{l}\mathsf{verifies}\ (i{:}\mathsf{heap})\ (p{:}\mathsf{pre})\ (q\ r{:}\mathsf{post}\ B) : \mathsf{Prop} := \\ \quad \exists h.\ (p * \mathsf{this}\ h)\ i \wedge \forall a\ f.\ (q\ a * \mathsf{delta}(\mathsf{this}\ h))\ i\ f \to r\ a\ i\ f\end{array}$$

and bind_pre, bind_post, read_pre, read_post abbreviate the STsep pre and postcondition for bind and read; e.g., $\mathsf{bind\_pre}\ p_1\ q_1\ p_2 = \lambda i.\ (p_1 * \mathsf{top})\ i \wedge \forall x\ f.\ (p_1 \multimap q_1)\ (\mathsf{Val}\ x)\ i\ f \to (p_2 * \mathsf{top})\ x\ f)$, and analogously for bind_post, read_pre and read_post.

One possible way to discharge this obligation is simply to unfold the abbreviations and attempt the proof directly. But, knowing that the corresponding program is $\mathsf{bind}\ (\mathsf{read}\ x)\ e$, a simpler strategy is as follows. First show that the read itself is safe; that is, $p_1\ i \to (x \hookrightarrow_A v)\ i$ holds for some v. Then show that $e\ v$ is safe to execute in the heap i, that is $p_1\ i \to \mathsf{verifies}\ i\ (p_2\ v)\ (q_2\ v)\ q_1$. This strategy can be codified and *verified* in the form of a lemma. We generalize with respect to the heap i, and make the hypothesis $p_1\ i$ implicit (since it persists across the subgoals) to obtain:

$$\begin{array}{l}\textbf{Lemma}\ \mathsf{eval\_bind\_read} : \\ \quad \forall x{:}\mathsf{loc}, v{:}A, p_2{:}A{\to}\mathsf{pre}, q_2{:}A{\to}\mathsf{post}\ B, i{:}\mathsf{heap}, r{:}\mathsf{post}\ B. \\ \qquad (x \hookrightarrow_A v)\ i \to \mathsf{verifies}\ i\ (p_2\ v)\ (q_2\ v)\ r \to \\ \qquad \mathsf{verifies}\ i\ (\mathsf{bind\_pre}\ (\mathsf{read\_pre}\ A\ x)\ (\mathsf{read\_post}\ A\ x)\ p_2) \\ \qquad\qquad (\mathsf{bind\_post}\ (\mathsf{read\_pre}\ A\ x)\ (\mathsf{read\_post}\ A\ x)\ p_2\ q_2)\ r\end{array}$$

We have proved similar lemmas for all possible interactions of STsep commands, and wrapped them into a tactic nextvc that pattern matches against the current proof state, and automatically chooses the appropriate lemma to apply. The nextvc tactic emits the immediate verification condition for the first effectful command in the proof state, and advances the proof state to the next command. This basically corresponds to symbolically evaluating the original program *in the logic itself*, discharging the verification conditions in the order of appearance.

In most related systems like PCC [33], ESC [25], Spec# [2], Sage [14] or WHY [13], the generator is an external program that is usually not verified itself. This is not to say that external generators have not been verified before. For example, several such projects have been carried out in Isabelle [48, 47]. These implement the verifier in the internal language of the prover, and then extract a corresponding ML program which can be compiled and executed. Our proposal seems much more direct and requires a smaller trusted computing base. Because condition generation is carried out by applying a lemma internally in Coq, we do not need to rely on the correctness of external tools that translate code back into Coq.

3. Interfaces and Modules

```
Record FiniteMap (K V : Type)
       (cmp : Π(k1 k2 : K). {k1 = k2} + {k1 ≠ k2}) : Type := {
  T : Type

  model : Type := list (K * V)

  rep : T → model → heap → Prop

  valid : T → heap → Prop := λt h. ∃m. rep t m h

  permutes : ∀t:T. ∀m1 m2:model. ∀h:heap.
      rep t m1 h → (Permutation m1 m2 ↔ rep t m2 h)

  distinct : ∀t:T. ∀m:model. ∀h:heap. rep t m h → distinct_keys m

  rep_precise : ∀t:T . precise (valid t)

  create : STsep emp T
      (λa i f. ∃t:T. a = Val t ∧ rep t nil f)

  destroy : Πt:T. STsep (valid t) unit(λa i f. a = Val tt ∧ emp f)

  insert : Πt:T. Πk:K. Πv:V.
      STsep (valid t) unit (λa i f. a = Val tt ∧
            ∀m:model. rep t m i → rep t ((k, v)::(remove cmp k m)) f)

  lookup : Πt:T. Πk:K.
      STsep (valid t) V ( λa i f. ∀m. rep t m i → rep t m f ∧
                          ((∃v:V. a = Val v ∧ assoc cmp k m = Some v) ∨
                           (a = Exn Failure ∧ assoc cmp k m = None)))
  delete : Πt:T. Πk:K.
      STsep (valid t) unit (λa i f. a = Val tt ∧
                  ∀m:model. rep t m i → rep t (remove cmp k m) f)
}.
```

Figure 3. Interface for a Finite Map

Thus far, we have seen how Ynot makes it possible to write imperative code in the style of Haskell, how principal specifications are inferred, how those specifications can be explicitly weakened by the programmer, and how STsep can be used to achieve separation-style, small-footprint specifications. In this section, we consider how these mechanisms can be combined with Coq's abstraction mechanisms to implement (first-class) abstract data types (ADTs).

We begin by considering an interface for imperative finite maps (Figure 3) which is intended to abstract implementation details. Usually, an interface for an ADT consists of an abstract type and type signatures for a set of operations over that type. These are conveniently packaged together in Coq as a dependent record type, or alternatively as a module. Here, we chose to use a record type, because then our ADT modules can be first-class.

In the setting of Ynot, the type signatures of ADT operations can capture not only types, but of course, specifications. However, it is important to make sure that the signatures are suitably abstract so that they admit different implementations. Therefore, in addition to abstracting an implementation type T, the interface abstracts over a representation predicate rep which associates an implementation value of type T to an idealized model m in a given state. Here, we have chosen to model finite maps as *functional* association lists, but effectively quotient the lists through the predicates permutes and distinct so that they behave like finite maps. distinct_keys is a predicate that holds if association lists do not have redundant keys. That is, in model, the order of the elements in the association list is immaterial, and we demand that keys occur at most once.

In the case of the *imperative* association list implementation, T will be instantiated with the concrete type loc and the rep predicate can be defined so that:

$$\mathsf{rep}\ t\ m\ h\ := \exists m'.\ \mathsf{distinct\_keys}\ m' \wedge \mathsf{Permutation}\ m\ m' \wedge (t \mapsto m')\ h$$

But of course, a different implementation will choose different definitions for T and rep as we show in Section 3.1 below.

The interfaces for the operations use the rep predicate to specify the effect of the operation on the abstract type in terms of the model. For example, when we insert key k with value v into a finite map t, then if t represents the functional association list m on input, then on output, t represents the list $(k, v) :: (\mathsf{remove\ cmp}\ k\ m)$. If we then lookup k, we can prove that we will get the value v back as a result, and all of the reasoning can be done in terms of the simple functional model. For all intents and purposes, clients of the FiniteMap interface can reason as if the implementation is a simple reference to this functional model, when in fact, the implementation can have drastically different internals.

When we move to interfaces such as this one, the use of the separation monad STsep becomes crucial. Recall that if we attempt to use ST, then to get sufficiently strong post-conditions, we must reveal which locations are *not* affected by a command. The easiest way to do this is to express the output heap as a function of the input heap, but doing so tends to reveal implementation details. Combining an abstract predicate, such as rep, with the small-footprint abstraction of STsep ensures that these details can remain safely hidden behind the interface, and yet clients can still effectively reason about state that is not "owned" by the abstraction.

Finally, our interface for finite maps requires a proof that the rep predicate is *precise* through the rep_precise component. A heap-predicate P is precise when for all heaps h, if P holds of some subheap of h, then that subheap is uniquely determined [36]. Precision is important for making inferences about the values stored in the same chunk of heap. For example, if we know that $(x \mapsto v_1)\ h$ and $(x \mapsto v_2)\ h$, we should be able to conclude that $v_1 = v_2$, because the location x can point to only one value in any given heap. This property is basically a more concrete statement of the fact that $x \mapsto -$ is a precise predicate. We have found in practice, clients of our finite map structures will need to make similar inferences about the values stored into the heap occupied by the finite map. Exposing the property that rep is precise makes such inferences possible, while at the same time manages to keep the actual implementation of rep hidden.

In summary, the fact that a Coq dependent record type allows us to abstract over types, predicates, and terms makes it easy to capture true ADT interfaces in a uniform fashion. The use of STsep in conjunction with an abstract representation predicate in terms of an idealized model is crucial for achieving natural, yet workable specifications.

3.1 An Alternative Implementation: Hashtables

We can easily build an implementation of the FiniteMap interface using the association list operations defined at the beginning of Section 2. In this section, we describe a more interesting implementation we have constructed, namely a hash table. Our library also includes an implementation based on splay-trees, but space precludes a detailed discussion of both, so we focus on the hash tables.

Our hash table module is a functor that is parametrized by the key and value types, hash and comparison functions for keys, and a natural number n where $n > 0$. The tables are represented using an array of pointers to n buckets, where the buckets are themselves implemented as finite maps. Thus, the hash table functor also abstracts over a FiniteMap module, which can be instantiated with, for instance, the association list module, our splay-tree module, or even a nested hash-table. Therefore, the HashTableFunctor has an interface that looks like this:

```
HashTableFunctor :
  Π(K V : Type)(hash : K → nat)
    (cmp : Π(k1 k2:K), {k1 = k2} + {k1 ≠ k2})
    (Bucket : FiniteMap K V cmp)
    (len : nat)(lengt0 : len > 0). FiniteMap K V cmp
```

Within the body of the functor definition, we represent the abstract type T as an array of len locations, each of which points to

a Bucket.T value. In Ynot, an array of length len is treated similar to a function from nat to loc, with the guarantee that each location is distinct. Thus, to read or write an array element, a programmer can simply use pointer arithmetic to get at the appropriate location, and then use the location read and write primitives. Of course, the programmer must be able to prove that the location is in the footprint of the computation in order to perform the read or write. The proof must include a deduction that the pointer arithmetic used to obtain the location is in bounds for the array. The actual code for the various operations is relatively simple. For example, the code for insert is as follows:

```
Program Definition insert (arr: array len) (k:K) (v:V) :=
  let p := array_plus arr ((hash k) mod len) in
    do (bucket ← read p;
        Bucket.insert bucket k v) _ .
```

The code begins by calculating a pointer p to the appropriate bucket using the hash function and pointer arithmetic on the array. We then dereference the pointer to get a *bucket* element of type Bucket.T. Finally, we invoke the Bucket.insert operation to insert the key and value into the *bucket*.

To verify that the code meets the FiniteMap interface, we must choose an appropriate definition for rep. We say a bucket b_j is a valid_bucket with respect to an index j, model m and heap h when:

1. $\exists m_j$, Bucket.rep b_j m_j h, and
2. $\forall (k, v) \in m_j, (k, v) \in m \wedge (\mathsf{hash}\ k) \bmod \mathsf{len} = j$, and
3. $\forall (k, v) \in m, (\mathsf{hash}\ k) \bmod \mathsf{len} = j$ implies $(k, v) \in m_j$.

That is, the bucket includes all and only the elements of the global model m whose keys map via the hash function to the index j.

We say that an index j is a valid_index with respect to an array arr, model m, and heap h when:

$$\mathsf{valid\_index}\ arr\ m\ j\ h := \exists b_j, (\mathsf{array\_plus}\ arr\ j \mapsto b_j * \mathsf{valid\_bucket}\ b_j\ m\ j)\ h$$

In English, each index of the array points to a valid bucket with respect to the model and the index. Finally, we say that an array arr represents an association list m in heap h when:

$$(\mathsf{valid\_index}\ arr\ m\ 0 * \cdots * \mathsf{valid\_index}\ arr\ m\ (\mathsf{len} - 1))\ h$$

This iterated separating conjunction can be coded in Coq with the use of an inductively defined predicate as follows:

```
Fixpoint iter_sep (n:nat)(P:nat→heap→Prop) {struct n} :=
 match n with
   | 0    ⇒ emp
   | S m ⇒ (P m) * (iter_sep m P)
 end.
```

and thus for the hash-table implementation, we define:

$$\mathsf{rep}\ arr\ m\ h\ :=\ \mathsf{distinct\_keys}\ m \wedge \mathsf{iter\_sep}\ \mathsf{len}\ (\mathsf{valid\_index}\ arr\ m)\ h$$

Given this definition for rep, the proofs that the operations meet their specifications are relatively straightforward, if tedious, and we refer the reader to the implementation for details. We do remark that the proof scripts are relatively large when compared to the code, in spite of our simplification tactics described in Section 2.3, and a library of tactics for reasoning about the separation connectives. The entire hash table implementation (BucketHashMap.v) consists of about 1200 lines (including comments and white-space). Of this, only about 300 lines are the actual definitions that make up the code; the rest consists of lemmas and proof scripts. Although we hope that some day, the proof burden will be much lower, we also believe that sophisticated invariants such as rep are far beyond what state-of-the-art decision procedures can synthesize or prove automatically.

4. Iterators

Figure 3 presents our basic finite map interface. In this section we discuss how to extend it with an iterator fold, which applies a computation c to each key and value pair in the map. In our case, the fold method must abstract over the return type of c as well as its *precondition and postcondition* so that we can capture the effectful behavior of the iteration. Without abstraction over types and predicates, it would not be possible to support such a general, higher-order operation.

To motivate the development, imagine that we want to implement a generic copy function that copies a collection of keys and values from one finite map implementation to another. Such a function would have the following type:

```
copy (F G : FiniteMap K V cmp) (t : F.T) :
    STsep (F.valid t) (G.T)
          (λa i f. ∃t' : G.T. a = Val t' ∧
                   ∀m:model. F.rep t m i →
                     (F.rep t m * G.rep t' m) f)
```

It takes two possibly different finite map implementations F and G (say, one can be an implemented with hash tables, and the other with splay trees), and a pointer t:F.T to the map of kind F. The function returns a pointer to the G copy. The postcondition guarantees that the input map is semantically preserved (i.e., t points into a memory that implements the same map m in both the initial and the ending heap), that the G copy also implements the same map m, and that the two copies occupy disjoint chunks of memory.

Given an appropriate definition for fold, we should be able to implement the copy procedure as follows:

```
Definition copy (F G : FiniteMap K V cmp) (t : F.T) :=
  t' ← G.create;
  F.fold t tt (λk v b. G.insert t k v);
  ret t'.
```

Notice that to write copy as above, fold has to be able to take an effectful computation G.insert and iterate it across the finite map. But in order to conclude that at the end of the fold, we have produced a copy of the original map, we need the specification of fold to capture the *aggregate* effect of applying the computation to each key and value pair. Of course, the interface for fold needs to be sufficiently abstract that we can efficiently realize it for the different implementations.

Furthermore, it is desirable for the specification to prevent any changes to the map during iteration. While iterators that change the map are possible, there are situations that cause most implementations to break. For example, if a key is deleted from the map during iteration, should its associated value be passed to the computation? More subtley, in the context of the splay-tree iterator, a lookup operation can re-arrange the tree, making it difficult to track where the iterator should look for the next key and value. To avoid these conundrums, collection libraries, such as those found in Java, attempt to rule out modifications to the collection during iteration, but are forced to do so through a run-time check. This is usually accomplished by stamping the collection with a version number that is incremented each time the collection is changed. If the iterator detects a version mis-match, it will throw an exception. For example, Java collections throw the `ConcurrentModificationException` when they detect a version mis-match.

In Ynot, we do not need time-stamps or run-time checks to ensure the correctness of the iterator as the conflicts are simply

ruled out by the specification. Here then is our specification of fold.

```
fold (t:T) (b:B)
    (c : Πk:K. Πv:V. Πb′:B. STsep (P k v b′) B (Q k v b′)) :
STsep (λi:heap. ∃m:model. (rep t m ∗
          (λh. ∀m′. Permutation m m′ → fold_pre P Q m′ b h)) i)
      B
      (λa i f. ∃m:model. ∃ h:heap.
            (rep t m ∗ (λi′. fold_pre P Q m b i′ ∧ h = i′)) i ∧
            (rep t m ∗ fold_post Q m b a h) f)
```

The precondition requires that the initial heap can be split into the portion belonging to the map (which thus satisfies $\mathsf{rep}\, t\, m$) and some state h that is owned by the computation c. The portion h must satisfy an inductive predicate fold_pre, defined below. Intuitively, fold_pre characterizes a chain of heaps $h{=}h_1, h_2, \ldots, h_n$, where h_j is the result of running the computation c in state h_{j-1} on the j^{th} key-value pair. Additionally, h_{j-1} must satisfy P, and h_j will satisfy Q as required by the specification of c. Notice that fold is parametric in P and Q so that any computation c (that does not access the map) can be supplied for iteration. Also, note that the order that the keys and values are presented to the computation is non-deterministic: an implementation is free to present them in any order that is a permutation of the model.

The postcondition also uses an inductive predicate fold_post (defined below) that effectively calculates the final state in the sequence $h{=}h_1, h_2, \ldots, h_n$ described above. The postcondition also ensures that the model of the original map is preserved, as $\mathsf{rep}\, t\, m$.

Finally, we present the predicates fold_pre and fold_post which "iterate in the logic" the precondition and postcondition P and Q of c, to capture the requirements that fold makes on the state owned by c.

```
Fixpoint fold_pre (P:K→V→B→pre) (Q:K→V→B→post B)
                  (m:model) (b:B) (h:heap){struct m} : Prop :=
  match m with
    | nil ⇒ ⊤
    | ((k, v)::m′) ⇒ P k v b h ∧
        ∀f:heap. ∀a:B. Q k v b (Val a) → fold_pre P Q m′ a f
  end.
```

The predicate is defined by primitive recursion on the structure of the model m. When m is the empty list, c will not be executed at all, so we impose no preconditions on its state (proposition $\top$). When m contains a key-value pair (k, v), c will be executed at least once, and hence its state should satisfy $P\, k\, v\, b\, h$. If we get a value a and heap f out of the execution of c, we are guaranteed they satisfy $Q\, k\, v\, b\, (\mathsf{Val}\, a)$. But then this needs to guarantee we can iterate c over the remaining part of the finite map, and thus we must show it implies $\mathsf{fold\_pre}\, P\, Q\, m'\, a\, f$. In essence, this is a generalization of the precondition for binary sequential composition to n-ary sequential composition, where n is determined by the length of the model m.

Similarly, the predicate fold_post iterates in the logic the postcondition Q:

```
Fixpoint fold_post (Q:K→V→B→post B) (m:model) (b:B)
                   (a:ans B) (i f:heap) {struct m} : Prop :=
  match m with
    | nil ⇒ i = f ∧ a = Val b
    | ((k, v)::m′) ⇒ (∃b′:B. ∃h:heap. Q k v b (Val b′) i h ∧
                        fold_post Q m′ b′ a h f) ∨
                     (∃e:exn, a = Exn e ∧ Q k v b a i f)
  end.
```

Notice how it allows the client to reason about the iteration and the state of the finite map even if c throws an exception. Again, this predicate generalizes the postcondition of a binary sequential composition to an n-ary case.

We can now show an implementation of an iterator for the specific instance of hash tables. We do note here that we have implemented a similar iterator for association lists and splay trees as well, to confirm that the specification we give above is general enough to support at least three different implementations.

For the purposes of specification of the loop invariants (which we omit here, but present in the Coq implementation), the hash table iterator is split into three functions as follows. First we have a function fold_bucketn which takes an index into the array of bucket pointers and folds the function over the bucket with the particular index. We make the same assumptions as in Section 3.1. For example, *arr* is an array of len locations, each of which points to a Bucket, keys have type K, values have type V, etc.

```
Program Definition
fold_bucketn (arr: array len) (j:nat) (pf:j < len) (b:B)
             (c:Πk:K. Πv:V. Πb:B. STsep (P k v b) B (Q k v b)) :=
    do ( bucket ← array_plus arr j ;
         Bucket. fold bucket b c) _ .
```

The next function is fold_bucketndown. It traverses the bucket array *up to the index* j, calling fold_bucketn over each bucket encountered. The traversal is implemented via Coq's primitive recursion on the index j. Notice the use of Coq wildcards to avoid writing the proofs within the program that the bucket indices are within bound.

```
Fixpoint fold_bucketdown (arr: array len) (j:nat) (pf:j < len) (b:B)
             (c:Πk:K. Πv:V. Πb′:B. STsep (P k v b′) B (Q k v b′)) :=
match j with
    | 0   ⇒ do (ret b) _
    | S k ⇒ do ( r ← fold_bucketdown arr k _ b c ;
                 fold_bucketn arr k _ r c) _
  end.
```

Finally, the actual implementation of fold is simply a wrapper, immediately calling fold_bucketdown, passing the size of the bucket array along.

```
Program Definition
fold (arr: array len) (b:B)
     (c : Πk:K. Πv:V. Πb′:B. STsep (P k v b′) B (Q k v b′)) :=
  do (fold_bucketndown arr len _ b c) _ .
```

5. Memoization

We close our examples by discussing an interesting use of *dependent* finite maps, namely verified memoization. A dependent finite map has the same interface as the one given in Figure 3, except that the value type V can depend upon a key as in:

```
Record DepFiniteMap (K : Type) (cmp : ···) (V : K → Type) := {
  T : Type
  model : Type := list ({k : K, v : V k})
  ···
  insert : Πt:T. Πk:K. Πv:(V k). STsep ···
  lookup : Πt:T. Πk:K. STsep (valid t) (V k) (···)
  ···
}.
```

Somewhat surprisingly, our code for all three finite-map implementations easily generalizes to this more expressive interface, and our implementation supports this. The advantage of a dependent finite map is that we can use it to approximate (and thus memoize) a function. Given a function $F : \Pi k{:}\mathsf{K}.\, \mathsf{V}\, k$, we can use a subset type for the value of the finite map, $\lambda k{:}\mathsf{K}.\, \{x{:}\mathsf{V}\, k \mid F\, k = x\}$. For example, the following Ynot code is intended to compute the n^{th} Fibonnaci number. Let Memopad be the type of finite maps that can only store Fibonnaci numbers, i.e. $\mathsf{Memopad} = \mathsf{DepFiniteMap\ nat\ cmp}\ (\lambda k.\, \{x{:}\mathsf{nat} \mid \mathsf{ffib}\, k = x\})$, where ffib is a functional specification of the Fibonnaci function. The following code is lifted from one of our example memoization files and com-

putes the n^{th} Fibonnaci number:

```
Fixpoint ifib_mem(n : nat)(mem:Memopad)(t : mem. T){struct n}
    : STsep  (mem. valid t)({x : nat | ffib n = x})
             (λ a i f. mem. valid t f ∧ ∃v. a = Val v) :=
  let ifib_rec := λk. applyd t (ifib_mem k mem t) in
  match n with
    | 0    ⇒ do (ret 0)
    | S k  ⇒ match k with
        | 0    ⇒ do (ret 1)
        | S j  ⇒ do (n1 ← ifib_rec k ;
                    n2 ← ifib_rec j ;
                    ret (n1 + n2))
      end
  end.

Definition ifib(n : nat)
  : STsep  emp ({x : nat | ffib n = x})
           (λ a i f.emp f ∧ ∃v. a = Val v) :=
    call_mem _ (ifib_mem n) (size := 1 + n).
```

The entry point ifib simply invokes a memoization library, call_mem. The call_mem operation builds a memo table, which is a dependent finite map of type Memopad intended to associate n to ffib n. The memo table is passed to the ifib_mem function so that it can be used to cut-off re-computation. In particular, the applyd function first tries to find a value associated with k in the table t. Failing that, it invokes the recursive computation ifib_mem k and caches as well as returns the result. Finally, call_mem will deallocate the memo table after the computation is complete. If the memoization library is constructed with the hash table functor, then in principle, the computation should run in time proportional to n.

Notice that the return type of ifib n guarantees that a return value must be ffib n through the use of a dependent type. Yet the proof that the code is well-typed (i.e., respects the specification) is trivial. This is because the hard parts of the proof were localized into the correctness of the finite map implementations.

6. Compiling Ynot

We have developed a compiler for Ynot that produces object code compatible with GHC-compiled Haskell modules. The compiler works in three stages. First, code is extracted from Coq using a small Coq extension that eliminates functors and modules and normalizes all of the CiC terms. Second, the CiC terms are read in to the Ynot compiler and lowered to an explicitly-phased intermediate language. In the second stage, we eliminate proof terms and compile away inductives and pattern matching. Finally, we generate a set of GHC internal core language terms and compile them using the GHC code generator. Compiling through GHC's internal language allows us to ignore the surface syntax and type system of Haskell, while still getting the benefits of GHC's optimizer and run-time system[1]. In addition, since many Ynot terms have computationally irrelevant sub-terms, the laziness of Haskell is a benefit [9].

The Coq system already has a code extraction mechanism [27]. However, the Haskell extraction, which we wanted to use both for improved performance and a more advanced run-time system, isn't able to handle the entire Coq language. Also, we are eager to do experiments with much more aggressive optimization and code elimination techniques, and we needed an intermediate representation that preserves all of the Ynot typing information. The Ynot compiler is still in the early stages of development, and is one area of future work.

[1] Many of the Ynot terms cannot be typed by GHC's internal type system, since GHC's type system is too weak, but so far this has not been a problem.

7. Related work

Extended static and dynamic checking. An alternative to starting with a dependent type theory is to start with a standard effectful programming language, and attempt to retrofit facilities for expressing models, specifications, dependency, refinement, and proofs. For example, JML [7] and Spec# [2] extend Java and C# respectively with support for Hoare-style pre- and postconditions as well as object invariants and other features used to capture safety and correctness requirements of code. Then an automated SMT-style prover is used to try to discharge proof obligations and establish that the code meets its specification. While there have been great advances in automated prover technology over the past few years, these systems still have trouble discharging all but the most "shallow" specifications such as array bound checks and NULL-pointer checks, but their capabilities fall well short of full correctness. In contrast, dependent type theories make it possible to utilize automated deduction techniques, but when they fall short, also allow the programmer to construct explicit proofs of "deep" properties of programs. We claim that, for programs where deep reasoning matters, programmers will play an integral role in co-developing models, specifications, programs, and proofs.

Furthermore, the modeling and specification languages used by JML and Spec# are too weak to support a truly modular development of programs and specifications. For example, neither language has the expressive power to write and prove correct a principal specification for a higher-order iterator, such as our fold construct.

A related approach to refinements is the "design-by-contract" idea, recently implemented by Sage [14] and Deputy [8]. Here, programmers write boolean expressions as pre- and postconditions that are intended to be executed at run-time. Program analysis can optimize the run-time checks away, but in practice the overhead may still be significant. A more important issue is that the contracts must only contain benign effects in order to safely optimize them away without changing the behavior of the program. This becomes particularly important in the presence of concurrency, when contracts must avoid deadlock and racing over shared data. However, if we enforce that the boolean functions are benign according to some syntactic or typing criteria, then in practice, we cannot always use existing or efficient code for the contract. As a simple example, consider a splay-tree lookup which happens to re-balance the tree: The effects are benign (because eliminating the lookup is safe) but this seems to demand a deep proof.

Dependent types for programming. There is another emerging class of projects that are attempting to retrofit dependent typing to effectful programming languages based on indexed or phase-split dependent types. Notable examples include DML [50] and more recent ATS [49], Omega [41], Concoqtion [15], RSP1 [46] and Liquid Types [39]. These systems have a strong separation between the programming language (which may include effects) and the specification languages (which do not). For example, in Concoqtion, the programming language is based on OCaml, but the specification language is based on Coq. Coq terms can be used to index ML type constructors that classify values, so it becomes possible, for instance, to use the types to capture correctness requirements. However, neither of these systems can use indexed types to suitably capture and reason about effects themselves (e.g., updates to a mutable store.) Thus, while programmers can write effectful code, they can only reason completely about the pure subset.

Higher-order Hoare Logics, models, and implementations. There are a number of recent extensions of Hoare and Separation Logic to higher-order functional languages. Honda et al. [19] give a total correctness logic for PCF with references, and Krishnaswami et al. [23] formalize higher-order Separation Logic for a higher-order language and prove the correctness of a subject-observer pattern.

Weber [45] implements a first-order separation logic for a simple while-language in Isabelle/HOL and verifies an in-place list reversal algorithm. Preoteasa [38] implements a first-order separation logic in PVS for a language with recursive procedures and proves the soundness of the frame rule. Varming and Birkedal [43] implement in Isabelle a higher-order Separation Logic for an imperative language with simple procedures and prove the correctness of Cheney's copying garbage collector. None of the above languages considers the higher-order features such as polymorphism, quantification over constructors and type equalities, that we consider in Ynot. Yet as we have shown here, these are essential for modularity and realistic programming.

The approaches also differ from Ynot in the treatment of programs with errors (e.g., a dereference of a dangling pointer). They usually start with the universal domain of programs, out of which specifications carves subsets, according to the pre- and postcondition. But, since erring programs are part of the universal domain, the semantics has to make provisions for their treatment. This is not so in Ynot where well-typed programs do not produce errors, and thus errors do not show up during verification.

Another related approach is the refinement calculus [1] which has recently been adapted to Coq [6]. Here, correctness is established by showing that one program refines another: the abstract program serves as specification for the concrete one. In Ynot, specifications are types, rather than refinement relations, and thus the same distinction from above applies. Another key difference is that the the refinement semantics directly interprets programs as predicate transformers in Coq. Therefore, the approach does not scale to include general higher-order functions, nor higher-order store [6, Sections 6.5-6.6].

Allowing for these features is the main reason why we implemented Ynot as an axiomatic extension, rather than a definition within type theory. Of course, then we needed to show that it is sound to do so [32, 37].

8. Summary and future work

Ynot is a dependently typed programming language that safely extends Coq with stateful side-effects and a way to reason about them via a version of Hoare and Separation Logic.

Unlike in many other Hoare-style logics, in the partial correctness specifications are types, which leads to a distinctive and concise style of use whereby programming, specification and verification are all integrated and inter-dependent. The first example of integration is that our inference rules for partial correctness double as programming primitives in the monadic style of Haskell: monadic `bind` corresponds to sequential composition, monadic `return` to the rule of assignment, monadic `do` combines the rules of consequence and frame.

The biggest advantage brought by the integration is that programs, types, predicates and proofs can all *abstract over* other programs, types, predicates and proofs, which facilitates information hiding, code *and proof* reuse, and modularity in general. These abstraction features already existed in pure type theories, but to date were not reconciled with effects.

We illustrated the modularity features of Ynot by implementing a *certified* library for mutable data structures such as association lists, hash tables and splay trees, which are all instances of the finite map interface, and can freely be interchanged in the clients of the finite maps. We illustrated that Ynot can support the programming with and reasoning about important higher-order stateful patterns such as shared local state (e.g., the state storing the finite map is shared by the methods, but its details are hidden from the clients), higher-order iterators, and memoization. All of these essentially rely on the ability to write code that is polymorphic in types, but also in pre and postconditions.

In the most immediate future work, we plan to extend Ynot with other kinds of side effects, such as I/O, concurrency, higher-order control flow, and foreign functions. We also hope to improve support for equational reasoning about computations, and to provide better automation support for discharging verification conditions. Finally, we plan to further improve on the quality of the code generated by the Ynot compiler.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices, CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2004.

[3] N. Benton and A. Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, July 2001.

[4] N. Benton and U. Zarfaty. Formalizing and verifying semantic type soundness for a simple compiler. In *International Conference on Principles and Practice of Declarative Programming, PPDP'07*, pages 1–12, 2007.

[5] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *FOSSACS'07*, volume 4423 of *LNCS*, 2007.

[6] S. Boulmé. Intuitionistic refinement calculus. In *International Conference on Typed Lambda Calculus and Applications, TLCA'07*, pages 54–69, 2007.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[8] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2007.

[9] L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. In *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005. To appear.

[10] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Compaq Systems Research Center, Research Report 159, December 1998.

[11] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[12] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Conference on Programming Language Design and Implementation, PLDI'08*, page to appear, 2008.

[13] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[14] C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages, POPL'06*, pages 245–256, 2006.

[15] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: Indexed types now! In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'07*, 2007.

[16] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Conference on LISP and Functional Programming*, pages 28–38, 1986.

[17] G. Gonthier. A computer-checked proof of the Four Colour Theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2005.

[18] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming, ESOP'08*, page to appear, 2008.

[19] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Symposium on Logic in Computer Science, LICS'05*, pages 270–279, 2005.

[20] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, USENIX'02*, pages 275–288, Monterey, Canada, 2002.

[21] C. B. Jones. Some mistakes I have made and what I have learned from them. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 7–20. Springer-Verlag, 1998.

[22] T. Kleymann. Metatheory of verification calculi in LEGO: To what extent does syntax matter? In *Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*, pages 133–149, 1999.

[23] N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Modular verification of the subject-observer pattern via higher-order separation logic. Presented at the FTFJP 2007 workshop, 2007.

[24] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[25] K. R. M. Leino, G. Nelson, and J. B. Saxe. *ESC/Java User's Manual*. Compaq Systems Research Center, October 2000. Technical Note 2000-002.

[26] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages, POPL'06*, pages 42–54, 2006.

[27] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[28] N. Marti and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, page to appear, 2007.

[29] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[30] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science, LICS'89*, pages 14–23, Asilomar, California, 1989.

[31] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2007.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming, ICFP'06*, pages 62–73, Portland, Oregon, 2006.

[33] G. C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages, POPL'97*, pages 106–119, Paris, January 1997.

[34] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Symposium on Principles of Programming Languages, POPL'06*, pages 320–333, Charleston, South Carolina, January 2006.

[35] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic, CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[36] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Symposium on Principles of Programming Languages, POPL'04*, pages 268–280, 2004.

[37] R. L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare Type Theory. In *European Symposium on Programming, ESOP'08*, 2008.

[38] V. Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In *14th International Symposium on Formal Methods*, pages 508–523, August 2006.

[39] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Conference on Programming Language Design and Implementation, PLDI'08*, page to appear, 2008.

[40] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, January 2005.

[41] T. Sheard. Languages of the future. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04*, pages 116–119, 2004.

[42] M. Sozeau. Program-ing finger trees in Coq. In *International Conference on Functional Programming, ICFP'07*, pages 13–24, 2007.

[43] C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. In *Submitted for publication*, 2008.

[44] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming, ICFP'98*, pages 63–74, Baltimore, Maryland, 1998.

[45] T. Weber. Towards mechanized program verification with separation logic. In *Proceedings of CSL'04*, volume 3210 of *LNCS*, pages 250–264. Springer, 2004.

[46] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *International Conference on Functional Programming, ICFP'05*, pages 268–279, 2005.

[47] M. Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Institut für Informatik, Technische Universität München, 2005.

[48] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Applications of Higher Order Logic Theorem Proving, TPHOL'04*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320, 2004.

[49] H. Xi. Applied Type System (extended abstract). In *TYPES'03*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[50] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation, PLDI'98*, pages 249–257, Montreal, Canada, 1998.