Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant

Anonymous

Abstract

We present Fiat, a library for the Coq proof assistant supporting refinement of declarative specifications into efficient functional programs with a high degree of automation. Each refinement process leaves a proof trail, checkable by the normal Coq kernel, justifying its soundness. We focus on the synthesis of abstract data types that package methods with private data. We demonstrate the utility of our framework by applying it to the synthesis of *query structures* – abstract data types with SQL-like query and insert operations. Fiat includes a library for writing specifications of query structures in SQL-inspired notation, expressing operations over relations (tables) in terms of mathematical sets. This library includes a set of tactics for automating the refinement of these specifications into efficient, correct-by-construction OCaml code. Using these tactics, a programmer can generate such an implementation completely automatically by only specifying the equivalent of SQL indexes, data structures capturing useful views of the abstract data. We conclude by speculating on the new programming modularity possibilities enabled by an automated refinement system with proved-correct rules.

> *"Every block of stone has a statue inside it and it is the task of the sculptor to discover it."*
>
> — Michelangelo

1. Introduction

Programmers are accustomed to ensuring that their code's abstractions hold by using languages that *enforce modularity*, as between a data-structure class A and some of its client code B. The programmer can safely replace A with any data structure implementing a common interface, with the language guaranteeing that bugs in either implementation will not be able to modify the orthogonal state of B and break associated invariants. Another mode of abstraction is to use a *compiler* A to optimize the code of B in an intensional way (i.e., by inspecting syntax trees), with the assumption that the resulting code has the same behavior as B. Programmers are used to this decomposition, too, but they are also used to living without guarantees about the correct operation of compilers. What if a programming language could provide *enforced modularity* features guaranteeing the soundness of compilers, which we would give first-class status within the language? Then the programmer could decompose a program into a high-level specification of its *functionality* and semantics-preserving *optimizations* that produce efficient, executable programs. In such a language, programmers could confidently modify the optimizations to suit their performance requirements, in much the same way that they may swap hash-table and search-tree implementations of a dictionary interface.

Considering this idea more carefully, we can spot opportunities to go beyond the programmer's usual relationship with her compilers. Compilers generally do not play too well with each other. One feeds a compiler a whole program, and any optimizations that the compiler does not know about will go unapplied in the final code. If we make it lightweight enough to *compose* libraries of compilers within a program, then different programmers may independently implement different optimizations without worrying about breaking each other's code, if our language's enforced modularity is structured properly. This structure enables a programmer to take an existing compiler library and safely extend it with any optimizations specific to the domain of her program, with the language ensuring the soundness of any program optimized with the extended library.

This paper introduces **Fiat**, a system realizing the model we have just sketched. Fiat is a library for the Coq proof assistant, and mechanized proof-checking is at the heart of how it enforces modularity. First-class compilers are *optimization scripts* that transform programs in correctness-preserving ways, possibly resolving nondeterminism. Optimization scripts are Coq proof scripts proving refinement theorems, with the usual guarantee that Coq will reject any unsound proof. We can build up optimization-script libraries full of what one may think of as first-class compilers. Our main case study to date involves transformations closer to the idea of *query planning* in the database community, but we think of query planners as just a special sort of compiler. We start with declarative queries over relational tables, transforming them into efficient, correct-by-construction OCaml code. We have implemented "domain compilers" at varying levels of automation; we can do completely automated planning for a common class of queries, and with more work the programmer can apply some of our more advanced strategies for choosing data structures or algorithms.

Figure 1 shows an example Fiat implementation, for a simple data structure representing a book store. We consider this example in more detail later in the paper, but for now we just want to give a basic sense of how Fiat may be used. The code excerpt begins with a definition of a data type as a set of methods over a relational database schema (whose definition we give later, in Figure 8). Methods are written in SQL-inspired syntax, saying more of what we want computed than how to compute it. These expressions have rigorous meanings in Coq, standing for mathematical sets. For each of our two database tables, we define an *index*, a data structure useful to look up entries by values of certain keys. The two Definitions of IndexFor values accomplish that purpose in the figure. Next, we define an *abstraction relation*, explaining how we propose to implement set-based relations using the concrete indexes

```
Definition BookstoreSpec := QueryADTRep BookstoreSchema {
  const Init (_: unit) : rep := empty,

  update AddBook (book: Book) : bool :=
    Insert book into Books,

  update PlaceOrder (order: Order) : bool :=
    Insert order into Orders,

  query GetTitles (author: string) : list string :=
    For (b in Books)
    Where (author = b!Author)
    Return (b!Title),

  query NumOrders (author: string) : nat :=
    Count (For (o in Orders) (b in Books)
          Where (author = b!Author)
          Where (b!ISBN = o!ISBN)
          Return ())
}.

Definition BookStorage : IndexFor Book.
  mkIndex [ BookSchema/Author; BookSchema/ISBN ].
Defined.

Definition OrderStorage : IndexFor Order.
  mkIndex [ OrderSchema/ISBN ].
Defined.

Definition Bookstore_AbsR abs
(conc : BookStorage * OrderStorage) :=
    abs!Books ≃ benumerate (fst conc)
  ∧ abs!Orders ≃ benumerate (snd conc).

Definition Bookstore : Sharpened BookstoreSpec.
  plan Bookstore_AbsR.
  finish sharpening.
Defined.
```

Figure 1. Specification for a bookstore ADT

we have defined. Finally, we begin an *optimization script* to generate a correct-by-construction implementation (shown later in Figure 11). Most of the work is done by a "domain compiler" called **plan**, which knows how to use indexes to implement queries efficiently. The programmer is free to chain sequences of library invocations and more manual steps, as needed to meet his performance target. Any sequence accepted by Coq is guaranteed to preserve correctness, just as in conventional programming any sequence of calls to an encapsulated data type is guaranteed to preserve its invariants.

Fiat works in the tradition of *program synthesis by stepwise refinement*. One starts with a very underconstrained nondeterministic program, which may be quite nonobvious how to execute efficiently. Step by step, the developer applies *refinement* rules, which replace program subterms with others that are "at least as deterministic," introducing no *new* behaviors beyond those of the terms they replace. Eventually, the program has been refined to a completely deterministic form, ideally employing efficient data structures and algorithms. So long as the library of primitive refinement steps is sound, we know that the final program implements the original specification.

Systems like Specware [20] and its predecessors [2, 16] at the Kestrel Institute have enabled interactive synthesis by refinement since the 1970s. However, only recently has Specware supported any kind of *mechanized proof about refinement correctness*, by instrumenting refinement primitives to generate Isabelle/HOL proof scripts. Specware also follows a very *manual model of choosing refinement steps*, which is understandable given the challenging problems it has been used to tackle, in domains like artificial intelligence and programming-language runtime systems. With Fiat, we instead focus (for now, at least) on more modest programming tasks, like those faced by mainstream Web application developers interacting with persistent data stores. We aim to replicate the automation level of traditional SQL systems, while adding high assurance about the correctness of refinement derivations, by carrying them out inside of Coq with full proofs. At the same time, we have designed the framework to allow seamless integration with manual derivations where they are called for, without weakening the formal guarantees.

This paper proposes *optimization scripts* as a new approach to modularly structure programs. Instead of just decomposing a program into "data structures + algorithms," we may also decompose into "functionality + optimizations," with a similar kind of *enforced modularity* to what we are used to with the former. We propose the Fiat architecture for realizing that decomposition, introducing a few simple primitives that suffice to enable a wide variety of optimization strategies. There are further opportunities for abstraction and modularity within *functionality*, in the form of DSLs for specifications à la the SQL-inspired notations of Figure 1; and within *optimization scripts*, in the form of other domain compilers or more tactics for building more customized optimization scripts for the SQL domain.

Section 2 introduces Fiat's basic notions of *computations* and their *refinement*. Section 3 lifts these ideas to *abstract data types*, which expose private data through methods with specifications. The material up to this point forms the general framework that we propose to enable this new kind of programming modularity. Section 4 presents *query structures*, one application of that framework, to the setting of SQL-style operations on relational tables, covering the full spectrum from defining new notations for *functionality* to exporting *optimization-script* building blocks at varying levels of automation. We have used this machinery to generate correct-by-construction OCaml programs with good performance; Section 5 includes more detail. We close with more discussion of related work. The entire Fiat framework, including all the examples discussed in this paper, is included as an archive accompanying this submission, and can be built and run using the standard distribution of Coq 8.4pl2.

2. Computation

The synthesis process in Fiat is one of *refinement*: users start out with a high-level specification that can be implemented in any number of different ways, and iteratively refine the possible implementations until they have produced an executable (and hopefully efficient) implementation.

Specifications in Fiat are represented as *computations*, or sets of values satisfying some defining property. Figure 2 lists the three combinators Fiat uses to define these sets: **ret** builds a singleton set, set comprehension $\pmb{\{}\_ \mid \_\pmb{\}}$ "picks" an arbitrary element satisfying a characteristic property, and the "bind" combinator, "$\_ \leftarrow \_; \_$", combines two computations. Throughout the text we will use the notation $c \rightsquigarrow v \equiv v \in c$ to emphasize that computations denote sets of "computed" values.

$$\mathbf{ret}\ a \equiv \{a\} \qquad\qquad \pmb{\{}a \mid P\ a\pmb{\}} \equiv \{a \mid P\ a\}$$

$$x \leftarrow c_a; c_b(x) \equiv \{b \mid \exists a \in c_a.\ b \in c_b(a)\}$$

Figure 2. Computation combinators

Consider the following (particularly permissive) specification of an insert function for a cache represented as a list of key-value pairs:

$$\mathsf{insert}\ k\ v\ l \equiv\ \pmb{\{}l' \mid l' \subseteq [(k, v)] \cup l\ \pmb{\}}$$

This specification only requires that insert does not inject extraneous elements in the list. When a key is not included, the specification

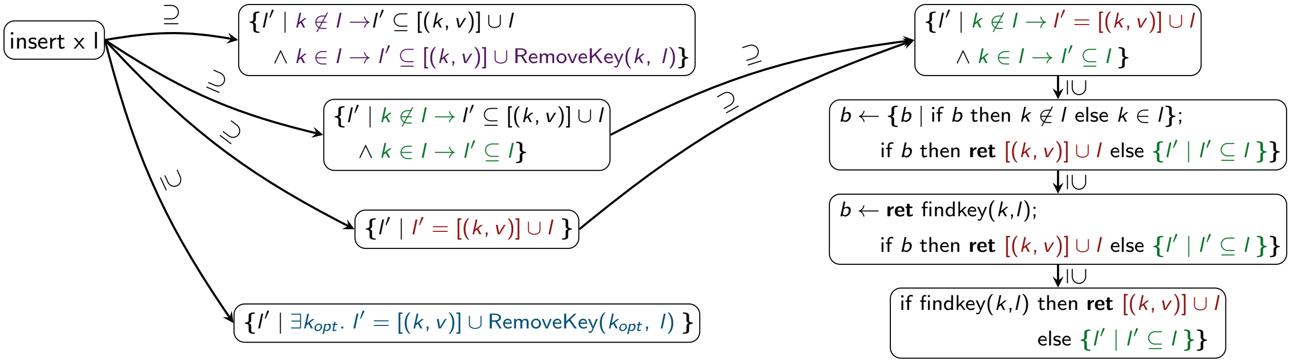

insert x l
$\supseteq$
$\{l' \mid k \notin l \rightarrow l' \subseteq [(k,v)] \cup l \wedge k \in l \rightarrow l' \subseteq [(k,v)] \cup \mathsf{RemoveKey}(k,\ l)\}$
$\supseteq$
$\{l' \mid k \notin l \rightarrow l' \subseteq [(k,v)] \cup l \wedge k \in l \rightarrow l' \subseteq l\}$
$\supseteq$
$\{l' \mid l' = [(k,v)] \cup l\}$
$\supseteq$
$\{l' \mid \exists k_{opt}.\ l' = [(k,v)] \cup \mathsf{RemoveKey}(k_{opt},\ l)\}$
$\supseteq$
$\supseteq$
$\{l' \mid k \notin l \rightarrow l' = [(k,v)] \cup l \wedge k \in l \rightarrow l' \subseteq l\}$
$\supseteq$
$b \leftarrow \{b \mid \text{if } b \text{ then } k \notin l \text{ else } k \in l\};$ if b then ret $[(k,v)] \cup l$ else $\{l' \mid l' \subseteq l\}\}$
$\supseteq$
$b \leftarrow$ ret findkey(k,l); if b then ret $[(k,v)] \cup l$ else $\{l' \mid l' \subseteq l\}\}$
$\supseteq$
if findkey(k,l) then ret $[(k,v)] \cup l$ else $\{l' \mid l' \subseteq l\}\}$

Figure 3. A space of possible refinements for the insert example

imposes no ordering on the result and somewhat counterintuitively does not require that the result include the new key-value pair. Such underspecification is not a bug. It allows for a wide range of caching behaviors: existing keys can be replaced or retained when they are reinserted, new keys can be inserted in an order that facilitates lookup, and old values can be dropped from the cache to maintain a constant memory footprint.

Each of these choices represents a more refined version of insert, with refinement defined by the subset relation $\subseteq$. Refinement forms a partial order on computations. Intuitively, a computation c' is a refinement of a computation c if c' only "computes" to values that c can "compute" to. Figure 3 shows a subset of the space of (non-deterministic) programs that we can explore through sequences of refinements from the initial definition of insert. The second column shows a number of alternatives reachable via initial refinement steps. The third column shows a further refinement combining two of these initial candidates, which ignores the insertion of duplicate keys and lets the cache grow unboundedly.

The first refinement in the third column applies a crucial combinator for decomposing a task into multiple easier ones: we can decompose insertion further into

$$b \leftarrow \{b \mid \mathsf{if}\ b\ \mathsf{then}\ k \notin l\ \mathsf{else}\ k \in l\}; \quad \mathsf{if}\ b\ \mathsf{then}\ \mathbf{ret}\ [(k,v)] \cup l\ \mathsf{else}\ \{l' \mid l' \subseteq l\}\}$$

where b is bound to the negation of a nondeterministic membership check for key k in list l. We consult b to decide whether to add the key-value pair to the list if k is not already used; or to nondeterministically shrink list l without introducing any new elements if k is already in the cache.

After this decomposition, it is natural to pick either the membership check or the list shrinking to refine further. A few basic properties of our computation combinators justify this sort of refinement of subterms. For instance, refinement may be pushed down through "bind" in two different ways:

$$\begin{aligned} c_a \subseteq c'_a &\longrightarrow \quad (x \leftarrow c_a; c_b(x)) \subseteq (x \leftarrow c'_a; c_b(x)) \\ (\forall x, c_b(x) \subseteq c'_b(x)) &\longrightarrow \quad (x \leftarrow c_a; c_b(x)) \subseteq (x \leftarrow c_a; c'_b(x)) \end{aligned}$$

These lemmas are the ingredients for decomposing the refinement process into more manageable chunks.

The usual monad laws [21] hold for computations under set equality $=$, allowing us to use them to simplify computations.

$$\begin{aligned} (x \leftarrow \mathbf{ret}\ a; c(x)) &= c(a) \\ (x \leftarrow c; \mathbf{ret}\ x) &= c \\ (y \leftarrow (x \leftarrow c_a;\ c_b(x));\ c_c(y) &= x \leftarrow c_a;\ y \leftarrow c_b(x);\ c_c(y) \end{aligned}$$

Fiat mechanizes derivation using Coq's setoid rewriting tactics, which extend Coq's rewriting machinery with support for partial-order relations other than Leibniz equality[1]. Given an appropriate set of refinement lemmas, implementing a given specification consists of applying rewrites until the term is reduced to a singleton computation (i.e., application of **ret**).

3. ADT Refinement

Fiat defines abstract data types [11] (ADTs) as records of state types and computations implementing operations over states. Figure 4 gives a specification of a cache as an abstract data type. In the CacheSig type signature, **rep** stands for an arbitrary abstract implementation type, to be threaded through the methods; this type placeholder has a special status in signatures. The CacheSpec functional specification is a *nondeterministic reference implementation* of a cache. That is, we choose a simple data representation type and its associated method implementations, in order to clearly express how any implementation of this ADT ought to behave. The representation type of CacheSpec does not even obviously lead to computable execution, since it is phrased in terms of mathematical sets. CacheSpec adds to our running example of insert a lookup *method* for retrieving values from the cache, an update method for updating keys already in the cache, and a *constructor* called empty for creating a fresh cache. We will fill in efficient data structures and algorithms through refinement.

```
ADTSig CacheSig :=
  empty : () → rep,
  insert : rep × Key × Value → rep,
  update : rep × Key × (Value → Value) → rep,
  lookup : rep × Key → Maybe Value

ADT CacheSpec implementing CacheSig :=
  rep := Set of (Key × Value)
  constructor empty := Return ∅
  method insert (r : rep, k : Key, v : Value) : rep :=
    {r' | ∀kv. kv ∈ r'  →  kv = (k, v)  ∨  kv ∈ r}
  method update (r : rep, k : Key, f : Value → Value) : rep :=
    {r' | ∀v. (k, v) ∈ r  →  ∀kv. kv ∈ r'  →  kv = (k, f(v))
                                          ∨ kv ∈ RemoveKey(k, r)}
  method lookup (r : rep, k : Key) : Maybe Value :=
    {v_opt | ∀v. v_opt = Some v  →  (k, v) ∈ r}
```

Figure 4. An abstract data type for caches

While mathematical sets are nice for specifying the Cache's methods, they are unsuitable for an implementation – we want to optimize the representation type as well. Fiat uses *abstraction relations*[9, 10] to enable representation type refinement. An abstraction

[1] The Coq documentation[6] has a full explanation of the machinery involved.

relation $\mathsf{A} \approx \mathsf{B}$ between two ADTs implementing a common signature ASig is a binary relation on the representation types A.**rep** and B.**rep** that is preserved by the operations specified in ASig. In other words, the operations of the two ADTs take "similar" input states to "similar" output states. Since operations in Fiat are implemented as computations, the methods of B may be computational refinements of A. Thus, an ADT method B.m is a refinement of A.m if

$$\begin{aligned}\mathsf{A.m} \simeq \mathsf{B.m} \;\; \equiv \;\; & \forall\, r_A\, r_B.\ r_A \approx r_B && \rightarrow \\ & \forall\, i\ r'_B\ o.\ \mathsf{B.m}(r_B,\ i) \leadsto (r'_B, o) && \rightarrow \\ & \exists r'_A.\ \mathsf{A.m}(r_A,\ i) \leadsto (r'_A, o) \wedge r'_A \approx r'_B\end{aligned}$$

The quantified variable i stands for the method's other inputs, beside the "self" value in the data type itself; and o is similarly the parts of the output value beside "self."

The statement of constructor refinement is similar:

$$\begin{aligned}\mathsf{A.c} \simeq \mathsf{B.c} \;\; \equiv \;\; & \forall\, i\ r'_B.\ \mathsf{B.m}(i) \leadsto r'_B && \rightarrow \\ & \exists r'_A.\ \mathsf{A.m}(i) \leadsto r'_A \ \wedge\ r'_A \approx r'_B\end{aligned}$$

B is a refinement of A if all the operations of B are refinements of the operations of A:

$$\mathsf{A} \simeq \mathsf{B} \;\; \equiv \;\; \forall \mathsf{o} \in \mathsf{ASig}.\ \mathsf{A.o} \simeq \mathsf{B.o}$$

To be completely formal, this relation $\simeq$ should be indexed by the abstraction relation $\approx$, so that we write $\mathsf{A} \simeq_{\approx} \mathsf{B}$ to indicate that relation $\approx$ demonstrates the compatibility of A and B. Then we can define more general refinement as:

$$\mathsf{A} \simeq \mathsf{B} \;\; \equiv \;\; \exists R.\ \mathsf{A} \simeq_R \mathsf{B}$$

Note that we may use any $\subseteq$ fact to rewrite within the code of any method, justified as an ADT refinement by picking simple equality as the abstraction relation. The transitivity of $\simeq$ justifies chaining such steps with others that make representation changes. Furthermore, it is always possible to build a default (very nondeterministic!) implementation for any abstraction relation:

$$\begin{aligned}\mathsf{default}^{\approx}_{\mathsf{m}}\ (\mathsf{A.m},\ r_B,\ i) \equiv \quad & \{(r'_B, o) \mid \forall r_A.\ r_A \approx r_B \rightarrow \\ & \exists r'_A.\ \mathsf{A.m}(r_A,\ i) \leadsto (r'_A, o)\ \wedge\ r'_A \approx r'_B\} \\ \mathsf{default}^{\approx}_{\mathsf{c}}\ (\mathsf{A.c},\ i) \equiv \quad & \{r'_B \mid \exists r'_A.\ \mathsf{A.c}(i) \leadsto r'_A\ \wedge\ r'_A \approx r'_B\}\end{aligned}$$

Thus, ADT refinement fits nicely into our existing approach of using setoid rewriting to refine computations: method and constructor refinement are straightforward, and refining a representation type is as simple as building an ADT with default implementations and continuing to refine from there. Since Fiat implements ADTs as records, once all the methods of an ADT are refined into fully deterministic computations, they can be lifted to functions that normal Coq can use without any special treatment.

3.1 Deductive Synthesis with Fiat

Armed with these definitions, a valid implementation of a reference ADT A can be expressed in Coq as an ADT B paired with a proof that it is a valid refinement:

$$\mathsf{Sharpened}\ A \;\; \equiv \;\; \Sigma\, \mathsf{B}.\ \mathsf{A} \simeq \mathsf{B}$$

In addition to the definitions making up the refinement framework discussed so far, Fiat includes a library of *honing tactics* for writing optimization scripts that develop Sharpened ADTs interactively[2]. Intuitively, one of these tactics applies a single refinement step to produce a refined ADT that can be Sharpened further. To see how ADT refinement works in Fiat, we will consider a derivation of an implementation of the cache specified by CacheSpec[3].

Figure 5 illustrates this derivation, with single-bordered boxes framing honing tactics and double-bordered boxes framing the goals they produce.

Honing Data Representations One of the key design choices when implementing a bounded cache is the policy used to evict entries from a full cache. Many selection algorithms depend on more information than the reference implementation provides. Conceptually, these algorithms associate an index with each key in the cache, which the insertion algorithm uses to select a key for eviction when the cache is full. By augmenting the reference type with an additional set holding the indexes that are assigned to active keys, we are able to refine toward a whole family of cache algorithms. The particular abstraction relation we will use is

$$r_v \approx_i (r'_v, r'_i) \;\equiv r'_v = r_v \wedge \forall k.\ (k, \_)\ \in\ r'_i \leftrightarrow (k, \_) \in r'_v$$

Fiat includes a honing tactic "hone representation using R" which, when given an abstraction relation R, soundly changes the refinement goal from Sharpened A to Sharpened A', where A' is an ADT with the new representation type and with methods and constructors built by the $\mathsf{default}_m$ and $\mathsf{default}_c$ functions, respectively. Applying hone representation to CacheSpec produces the Cache_2 ADT in Figure 5.

Honing Operations After honing the representation of CacheSpec, we can now specify the key replacement policy of a bounded cache as a refinement of the insert method. Fiat includes a honing tactic "hone method m" that generates a new subgoal for interactively refining m. Figure 5 shows the result of using "hone method insert" to sharpen Cache_2. We can refine the initial specification of insert in this goal to

$$\{l' \mid \exists k_{opt}.\ l' = [(k, v)] \cup \mathsf{RemoveKey}(k_{opt},\ l)\ \}$$

by rewriting with refine_ReplaceUsedKeyAdd per Figure 3. We then use a sequence of rewrites to select the key with the smallest index in r_i once r_v is full.

After selecting the replacement policy, the refined insert method still has dangling nondeterministic choices, with constraints like $\{r'_n \mid r'_n \approx r'_o\}$, arising from the $\mathsf{default}_o$ implementations. Determinizing these picks in every method amounts to selecting the particular replacement key policy. Initializing the index of a key to zero after insert and incrementing the index of a key after each lookup, for example, implements a least frequently used policy. Alternatively, we can use these indexes as logical timestamps by initializing the index of a key to a value greater than every index in the current map and having lookup leave indexes unchanged, implementing a least recently used policy. Once we have rewritten the goal further to select the latter policy, the "finish honing" tactic presents a the original Sharpened goal updated with the refined method body. Figure 6 presents the full optimization script for Figure 5.

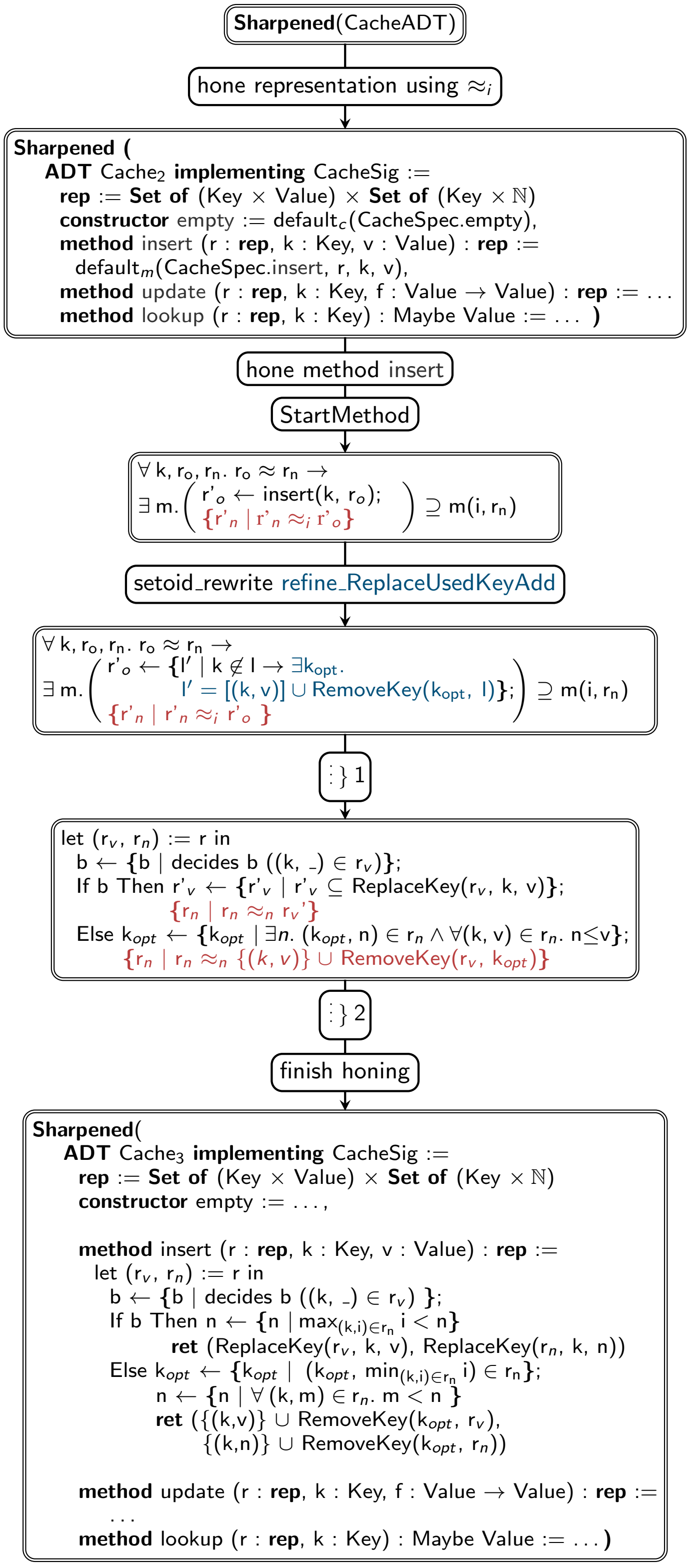
Invariants via Abstraction The idea of abstraction relations turns out to subsume the idea of *representation invariants* that must always hold on an ADT's state. Consider that an LRU implementation can "cache" the value of the greatest timestamp in its representation and use it in insert to select the next timestamp efficiently. In order to justify this refinement, however, we must maintain the invariant that this cached value holds the latest timestamp. To do so, we can include the invariant in the abstraction relation:

$$r_{vi} \approx_n (r'_{vi}, r'_n) \;\equiv r'_{vi} = r_{vi}\ \wedge\ r'_n = \max_{(k,i) \in r_{vi}} i$$

Now when honing insert, we can exploit the invariant on r'_n from $\approx_n$ to implement the selection of the timestamp. Since the abstraction relation formalizes the requirements on the saved index, the default implementation of insert automatically keeps the new piece of state up-to-date, albeit in a nondeterministic way that needs further refinement.

[2] As the keywords refine, replace, and change are already claimed by standard Coq tactics, Fiat uses the keyword hone in many of the tactics it provides – hence the name of the Sharpened predicate.

[3] The full Coq code for this example is in the accompanying archive.



```
Sharpened(CacheADT)

hone representation using ≈_i

Sharpened (
  ADT Cache₂ implementing CacheSig :=
    rep := Set of (Key × Value) × Set of (Key × ℕ)
    constructor empty := default_c(CacheSpec.empty),
    method insert (r : rep, k : Key, v : Value) : rep :=
      default_m(CacheSpec.insert, r, k, v),
    method update (r : rep, k : Key, f : Value → Value) : rep := . . .
    method lookup (r : rep, k : Key) : Maybe Value := . . . )

hone method insert

StartMethod

∀ k, r_o, r_n. r_o ≈ r_n →
∃ m. ( r'_o ← insert(k, r_o);
       {r'_n | r'_n ≈_i r'_o} )  ⊇ m(i, r_n)

setoid_rewrite refine_ReplaceUsedKeyAdd

∀ k, r_o, r_n. r_o ≈ r_n →
     ( r'_o ← {l' | k ∉ l → ∃k_opt.
∃ m.              l' = [(k, v)] ∪ RemoveKey(k_opt, l)};  ) ⊇ m(i, r_n)
       {r'_n | r'_n ≈_i r'_o }

⋮ } 1

let (r_v, r_n) := r in
  b ← {b | decides b ((k, _) ∈ r_v)};
  If b Then r'_v ← {r'_v | r'_v ⊆ ReplaceKey(r_v, k, v)};
            {r_n | r_n ≈_n r_v'}
  Else k_opt ← {k_opt | ∃n. (k_opt, n) ∈ r_n ∧ ∀(k, v) ∈ r_n. n≤v};
       {r_n | r_n ≈_n {(k, v)} ∪ RemoveKey(r_v, k_opt)}

⋮ } 2

finish honing

Sharpened(
  ADT Cache₃ implementing CacheSig :=
   rep := Set of (Key × Value) × Set of (Key × ℕ)
   constructor empty := . . . ,

   method insert (r : rep, k : Key, v : Value) : rep :=
     let (r_v, r_n) := r in
       b ← {b | decides b ((k, _) ∈ r_v) };
       If b Then n ← {n | max_(k,i)∈r_n i < n}
                ret (ReplaceKey(r_v, k, v), ReplaceKey(r_n, k, n))
       Else k_opt ← {k_opt | (k_opt, min_(k,i)∈r_n i) ∈ r_n};
             n ← {n | ∀ (k, m) ∈ r_n. m < n }
             ret ({(k,v)} ∪ RemoveKey(k_opt, r_v),
                   {(k,n)} ∪ RemoveKey(k_opt, r_n))

   method update (r : rep, k : Key, f : Value → Value) : rep :=
     . . .
   method lookup (r : rep, k : Key) : Maybe Value := . . . )
```

Figure 5. Derivation of an LRU implementation of CacheSpec

Implementing the Cache Once we are satisfied that the cache is sufficiently specified, we can implement it by honing the representation type to a data structure efficiently implementing the remaining nondeterministic operations on sets. For this example, we choose the finite maps implementation in Coq's standard library. We have proven a library of lemmas showing that the operations provided by finite maps correctly implement mathematical set operations used to specify the cache behaviors. The refinement lemmas can be used to implement a number of different high-level specifications. Conversely, since intermediate specifications (such as Cache$_3$) are independent from the implementation data structure, a library of refinement facts about a different data structure can be used to synthesize a different implementation. As an example, we could implement Cache$_3$ with a single map of keys to pairs of values and timestamps, or as two separate maps, one from keys to values and another from timestamps to keys. The former implementation uses less memory, but looking up the key to evict takes $O(n)$ time, whereas the latter can use the second map to do this lookup in $O(1)$ time.

Taking Stock We pause here to put the last two sections in context. Fiat provides a number of simple concepts for organizing the refinement of specifications into efficient code, where abstract data types are a central idea for packaging methods with private data. We apply refinements step-by-step with *optimization scripts*, which apply to Sharpened goals in Coq. These scripts are implemented in Ltac, Coq's Turing-complete tactic language, so it is possible to implement arbitrarily involved heuristics to choose good refinement sequences, packaged as callable tactic functions. The Coq kernel ensures that only semantically valid refinements will be admitted, providing a kind of *enforced modularity*. This idea is simple but powerful, supporting decomposition of programming tasks into uses of separate encapsulated components for *functionality* and *optimization*.

In the next section, we demonstrate the versatility of the approach for specifications inspired by SQL-style relational databases. We aim to convince the reader that Fiat's vocabulary of basic concepts matches well with the challenges that arise in optimization scripts that act like database query planners, and by extension in other natural functionality/optimization splits in programming.

4. Query Structures

This section presents a library for writing specifications of ADTs using reference implementations called *query structures* and with SQL-like query and insert operations. The library also provides tactic support for automatically refining those specifications into correct and efficient implementations, allowing users to generate

```
hone method insert.
 {
  StartMethod.
  setoid_rewrite refine_ReplaceUsedKeyAdd.
   setoid_rewrite refine_SubEnsembleInsert.          ⎫
   autorewrite with monad laws.                      ⎪
   setoid_rewrite refine_pick_KeyToBeReplaced_min.   ⎪
   setoid_rewrite refine_If_Then_Else_Bind.          ⎬ 1
   autorewrite with monad laws.                      ⎪
   setoid_rewrite refine_If_Opt_Then_Else_Bind.      ⎪
   autorewrite with monad laws.                      ⎭
   setoid_rewrite refine_pick_CacheADTwLogIndex_AbsR.  ⎫
   setoid_rewrite refine_pick_KVEnsembleInsertRemove   ⎪
                  with (1 := EquivKeys_H).             ⎪
   setoid_rewrite refine_pick_KVEnsembleInsert         ⎬ 2
                  with (1 := EquivKeys_H).             ⎪
   autorewrite with monad laws; simpl.                 ⎪
   finish honing. }                                    ⎭
```

Figure 6. Complete optimization script for Figure 5

$t \in \mathsf{Set}$ (* Types *)
$v \in \mathsf{V}$ (* Terms *)
$P \in \mathsf{Prop}$ (* Propositions *)
$i\ g\ h \in \mathsf{I}$ (* Identifiers *)

$H ::= \langle\overline{i :: t}\rangle$ (*Headings*)
$T ::= \langle\overline{i :: v}\rangle$ (*Tuples*)

$S ::=$ relation i schema H where P (*Relation Schemas*)
(*Query Structure Schemas*)
$Q ::=$ QueryStructure Schema $\overline{S}$ enforcing P

(*Functional Dependencies*)
attributes $\overline{i}$ depend on $\overline{g}$ $\quad\equiv\quad \forall t_1\ t_2\ .\ \overline{t_1!g = t_2!g} \rightarrow \overline{t_1!i = t_2!i}$
(*Foreign Keys*)
attribute i for g references h $\quad\equiv\quad \forall t_1 \in g\ .\ \exists t_2 \in h\ .\ t_1!i = t_2!i$

Figure 7. Syntax for reference query structure implementations

custom database-like ADTs. Figure 8 shows the definition of a schema describing a query structure for a simple book store ADT, which we used in Figure 1 from the introduction. This code is taken almost verbatim from the accompanying archive.

4.1 Specification of Query Structures

The Fiat query structure library includes a DSL for specifying reference query structures implemented in convenient notation, thanks to Coq's extensible parser. Figure 7 presents the grammar of this embedded DSL; the types t, terms v, and propositions P are exactly those of Coq itself. The grammar also draws on an infinite set of identifiers I.

Query structures are designed to align with the conceptual abstraction of SQL databases as sets of named *relations* (i.e. tables) containing *tuples*. The query structure library defines tuples as functions from *attributes* (i.e. column names) to values; the type of each value is described by a *heading* mapping attributes to types. Tuple projection is denoted using the ! notation:

$$\langle \text{Author::“T. Pynchon”}, \text{Title::“Bleeding Edge”}\rangle!\text{Title} \\ = \text{“Bleeding Edge”}$$

To align with the standard SQL notion of tables, relations are multisets (mathematical sets that can have repeated elements) of tuples. We implement that concept more concretely by pairing each tuple with a unique numeric index, storing these pairs as sets. Subsection 4.2 discusses how these indexes can be dropped for query structure implementations that only have SQL-like operations. *Relation (S) schemas* specify the heading of the tuples contained in a relation and also a set of constraints describing properties that included tuples must satisfy. Relations themselves are implemented as records, with a field for a mathematical set containing the relation's tuples, and a field for a proof that the schema constraints hold for every pair of tuples in the relation. A query structure is similarly described by a *query structure (Q) schema* that contains a set of schemas and a set of cross-relation constraints describing properties that must hold between tuples of different tables. A query structure is implemented as a record that contains a list of relations and proofs that each pair of relations satisfies its cross-relation constraints. The relations of a query structure are accessed using the ! notation: $qs!i$.

Data-Integrity Constraints The constraints contained in relation and query structure schemas are familiar to SQL programmers in the form of data-integrity properties like functional dependencies and foreign-key constraints. In our library, these constraints are simply representation invariants over the state of query structures, enforced by the proof fields of the relation and query structure records. Fiat's query structure library includes notations for these common SQL constraints (listed in Figure 7), but S and Q schema constraints are not limited to them: query structures can include arbitrary predicates over tuples. It is possible to specify that the population of a nation is always equal to the sum of the populations of its cities, for example. Why not simply omit these constraints and prove that our operations preserve them? Since query structures notations are simply an embedded DSL for writing ADTs, it is possible to write non-SQL-like operations for these ADTs – in addition to conforming with the standard data-integrity properties SQL programmers are familiar with, explicitly including these constraints allows users to be sure that they cannot write operations that go wrong.

```
Query Structure Schema
  [ relation Books has schema
      ⟨Author :: string, Title :: string, ISBN :: nat⟩
      where attributes [Title; Author] depend on [ISBN];
    relation Orders has schema
      ⟨ISBN :: nat, Date :: nat⟩ ]
  enforcing [ attribute ISBN for Orders references Books]
```

Figure 8. Query structure schema for a bookstore

$$\textbf{empty} \ \equiv\ \{q \mid \forall i \in qs.\ q!i = \emptyset\}$$

$$\textbf{For}\ b \ \equiv\ \mathit{result} \leftarrow b; \\ \{l \mid \textbf{Permutation}\ l\ \mathit{result}\}$$

$$(x\ \textbf{in}\ i)\ b \ \equiv\ \mathit{table} \leftarrow \{l \mid q!i\ \sim\ l\}; \\ \text{fold\_right}\ (\lambda a\, b\ .\ l \leftarrow a;\ l' \leftarrow b;\ \textbf{ret}\ (l \mathbin{+\!\!+} l')) \\ (\textbf{ret}\ [])\ (\text{map}\ (\lambda x.b)\ \mathit{table})$$

$$\textbf{Where}\ P\ b \ \equiv\ \{l \mid P \rightarrow b \rightsquigarrow l\ \wedge\ \neg P \rightarrow l = []\}$$

$$\textbf{Return}\ a \ \equiv\ \textbf{ret}\ [a]$$

$$\textbf{Count}\ b \ \equiv\ \mathit{results} \leftarrow b; \textbf{ret}\ \text{length}(\mathit{results})$$

Figure 9. Notations for initializing a query structure and defining query operations

4.2 Specification of Operations

The query structure library also provides a set of definitions and notations for specifying query and insertion operations mimicking standard SQL queries. Note that ADTs using query structures as reference implementations can support a mix of these operations and operations with arbitrary specifications, and can furthermore use SQL-style notations in the specifications of nonstandard operations. The most basic definition is **empty**, which returns a query structure containing only empty relations. All the operation specifications provided by the library have two implicit arguments: qs, the Q schema of the reference implementation; and q, the **rep** argument used by each method.

Querying Query Structures Figure 9 presents the notations the query structure library provides for specifying query operations. Queries specified by the **For** notation compute any permutation of a list of *result* tuples generated by a body expression. We use permutations in order to avoid fixing a result order in advance. Since queries are specified over relations implemented as mathematical sets, the definitions of these operations are a straightforward lifting of the standard interpretation of queries using comprehensions[3] and the list monad to handle computations. The **in** notation picks a *result* list that is equivalent ($\sim$) to the mathematical set of relation i; this is a placeholder for an enumeration method that is filled in by an implementation. This equivalence relation ignores the indexes of the tuples and only considers their multiplicity, and *result* thus disregards the indexes of the tuples in i. The body b is a function that

$$
\begin{array}{ll}
\mathsf{QuerySpec}(i,\ t,\ q') \ \equiv & \\
\text{Schema Constraints} \left\{ \begin{array}{l} \to i.P(t,\ t) \\ \to (\forall u \in q!i.\ i.P(u,\ t)) \\ \to (\forall u \in q!i.\ i.P(t,\ u)) \end{array} \right. & \\
\text{Query Structure Schema Constraints} \left\{ \begin{array}{l} \to (\forall g \neq i.\ qs.P_{i,g}(t,\ q!g)) \\ \to (\forall g \neq i, u \in q!g.\ qs.P_{g,i}(u,\ q'!i)) \end{array} \right. & \\
\qquad \to \forall u.\ u \in q'.i \ \leftrightarrow u \in (q.i \cup \{t\}) & \\
\qquad\quad \wedge\ (\forall g \neq i.\ q!g = q'!g) &
\end{array}
$$

$$
\begin{array}{l}
\textbf{Insert } t \textbf{ into } i \ \equiv \\
\quad id \leftarrow \{id \mid \forall u \in q!i.\ u.id \neq id\}; \\
\quad q' \leftarrow \{q' \mid \mathsf{QuerySpec}(i,\ (id,\ t),\ q')\}; \\
\quad b \leftarrow \{b \mid b = true \leftrightarrow (\forall u.\ u \in q'!i \ \leftrightarrow\ u \in (q \cup \{t\}))\}; \\
\quad \textbf{Return } (q', b)
\end{array}
$$

Figure 10. Notations for insert operations

is mapped over each tuple in *result*, producing a list of computations of query results for each element. This list of computations is then flattened into a single list of query results. Finally, **Where** clauses are allowed to use arbitrary predicates. Decision procedures for these predicates are left for an implementations to fill in.

Query Structure Inserts Whereas queries are *observers* for the query structure, insertion operations are *mutators* returning modified query structures, which by definition must satisfy the data-integrity constraints specified by their Q schema. The naive specification of insertion always inserts a tuple into a query structure:

$$\textbf{Insert } t \textbf{ into } i \ \equiv\ \{q' \mid \forall u.\ u \in q'!i \ \leftrightarrow u \in (q!i \cup \{t\})\}$$

This specification is unrealizable in general, however, as there does not exist a proof of consistency for a query structure containing a tuple violating its data-integrity constraints. The specification of **Insert** given in Figure 10 thus only specifies insertion behavior when the tuple satisfies both the S and Q schema constraints. This specification highlights Fiat's ability to specify method behavior at a high level without regard for implementation concerns. The specification delays the decision of how to handle conflicts (i.e. ignore the insertion, drop conflicting tuples, etc.) to subsequent refinements[4]. As we shall see in the following section, this specification can be automatically transformed into a more readily implementable form.

4.3 Honing Query Structures

Figure 1 contains the specification of an ADT using a query structure with the bookstore schema from Figure 8 and insert and query operations specified with the notations from the query structure library. As an initial step in implementing this specification, the library provides a fully automated tactic for removing the data-integrity constraints from the representation type of the ADT, building an ADT that uses *unconstrained* query structures, i.e. collections of mathematical sets with no proof components, freeing subsequent refinements from having to consider these proof terms. We pick an abstraction relation enforcing that a query structure is equivalent to an unconstrained query structure if the two contain equivalent sets of tuples:

$$q \approx q' \ \equiv\ \forall i \in qs.\ q!i = q'!i \qquad (1)$$

The implementation of this tactic is straightforward for the **empty** constructor. Queries over unconstrained query structures can also be constructed trivially if they are built from the notations in Figure 10, as those definitions do not reference the proof components of a query structure. For insertions, the tactic applies a lemma showing that if a tuple passes a series of consistency checks, it is possible to build the proof component of a query structure containing that tuple. By running these consistency checks before inserting a tuple into the unconstrained query structure, this lemma shows that there *exists* some query structure, i.e. that the abstraction relation is preserved. For a concrete query structure schema, these checks are materialized as a set of nondeterministic choices of decision procedures for the constraints. If there are no constraints, the tactic simplifies these checks away entirely. Figure 11 shows the result of applying this tactic to the bookstore ADT.

Refining the decision procedures for the remaining constraints into a concrete implementation is easily done by first transitioning to a query-based representation. In the foreign-key case, we refine

```
{b | b decides (∃ book∈ Books. book!ISBN = order!ISBN)}
```

into

```
c ← Count (For (book in Books)
             Where (book!ISBN = order!ISBN)
             Return ());
ret (c ≠ 0).
```

Similarly, functional-dependency checks are refined from

```
{b | b decides (∀ book'∈ Books.
                 book!ISBN = book'!ISBN →
                 book!Author = book'!Author ∧
                 book!Title = book'!Title)}
```

into

```
c ← Count(For (book' in Books)
            Where (book!ISBN = book'!ISBN)
            Where (book!Author ≠ book'!Author ∨
                   book!Title ≠ book'!Title)
            Return ());
ret (c = 0)
```

This type of refinement allows clients of the library to reuse the existing query machinery to implement these checks, and facilitate the automatic derivation of efficient query plans.

Importantly, this tactic only removes constraints from operations built from the notations provided by the library – any "exotic" operations will have to be refined manually to show that they preserve (1). Having removed the constraints automatically for "standard" queries, we now consider how to construct data structures that efficiently implement query and insertion operations.

4.4 The Bag Interface

The data structures used to store and retrieve data records are created, accessed, and modified through a unified Bag interface, which guarantees that all underlying implementations behave as multi-sets.This interface is implemented as a Coq type class parametrized over three types: TContainer, the type of the underlying representation (e.g. lists); TItem, the type of the items stored in the bag (e.g. tuples); and TSearchTerm, the type of the search terms used to look up items matching specific conditions (e.g. a function mapping tuples to Booleans). These types are augmented with a minimal set of operations whose behavior is described by a small number of consistency properties. While maintaining sufficient expressive power to allow for efficient retrieval of information, keeping the interface reduced makes it relatively simple to implement new instances.

The bag interface is split between constants, methods, and axioms. Methods operate on containers, and include benumerate,

[4] Note that regardless of how conflicts are resolved, the synthesis process ensures that any result must be equivalent to *some* query structure satisfying the data-integrity constraints of the schema.

```
ADTRep (UnConstrQueryStructure BookStoreSchema) {
  const Init (_: ()) : rep :=
    ret (DropQSConstraints (QSEmptySpec BookStoreSchema)),

  meth PlaceOrder (u: rep , order: Order) : bool :=
    id ← {id | ∀ order'∈ u!Orders. order'!id ≠ id};
    b ← {b | b decides (∃ book∈ Books.
                          book!ISBN = order!ISBN)};
    ret if b then (u!Books, u!Orders ∪ {(order, id)}, true)
             else (u!Books, u!Orders, false),

  meth AddBook (u: rep , book: Book) : bool :=
    id ← {id | ∀ book'∈ u!Books. book'!id ≠ id};
    b ← {b | b decides (∀ book'∈ Books.
                          book!ISBN = book'!ISBN →
                          book!Author = book'!Author ∧
                          book!Title = book'!Title)}
    ret if b then (u!Books ∪ {(book, id)}, u!Orders, true)
             else (u!Books, u!Orders, false),

  meth GetTitles (u: rep , author: string) : list string :=
    titles ← For (b in u!Books)
             Where (author = b!Author)
             Return b!Title;
    ret (u, titles),

  meth NumOrders (u: rep , author: string) : nat :=
    num ← Count (For (b in u!Books) (o in u!Orders)
                 Where (author = b!Author)
                 Where (b!ISBN = o!ISBN)
                 Return ());
    ret (u, num) }
```

Figure 11. The bookstore ADT after removing data-integrity proofs

which returns a list of all items stored in a container; binsert, which returns a copy of a container augmented with one extra item; and bfind, which returns all items matching a certain search term. Axioms specify the behavior of these transformations, in relation to two constants: bempty, the empty bag; and bfind_matcher, a matching function used to specify the behavior of bfind: calling bfind on a container must return the same results, modulo permutation, as filtering the container's elements using the bfind_matcher function (including this function allows us to retain maximal generality by allowing different bag instances to provide widely varying types of find functions). Finally, for performance reasons, a bag implementation is required to provide a bcount method (elided here), used to count elements matching a given search term instead of enumerating them.

```
Class Bag (TContainer TItem TSearchTerm : Type) := {
  bempty : TContainer;
  bfind_matcher : TSearchTerm → TItem → bool;

  benumerate : TContainer → list TItem;
  bfind : TContainer → TSearchTerm → list TItem;
  binsert : TContainer → TItem → TContainer;

  binsert_enumerate :
    ∀ inserted container,
      Permutation
        (benumerate (binsert container inserted))
        (inserted :: benumerate container);
  benumerate_empty : benumerate bempty = [];
  bfind_correct :
    ∀ container search_term,
      Permutation (filter (bfind_matcher search_term)
                          (benumerate container))
                  (bfind container search_term)}.
```

4.5 Bag Implementations

Fiat provides two instances of the Bag interface, one based on lists and the other based on AVL trees through Coq's finite map interface. The definitions making up the list instance are extremely simple and can thus be reproduced in their entirety below:

```
Instance ListAsBag
        (TItem TSearchTerm: Type)
        (matcher : TSearchTerm → TItem → bool) :
Bag (list TItem) TItem TSearchTerm := {|
    bempty := nil;
    bfind_matcher := matcher;

    benumerate container := container;
    bfind container search_term := filter (matcher search_term)
          container;
    binsert container item := item :: container |}.
```

To speed up counting operations, we also augmented the list-based implementation with a length field.

The tree-based implementation, though lengthier, is also readily explained: it organizes elements of a data set by extracting a key from each element, and grouping elements that share the same key into smaller bags. The smaller bags are then placed in a map-like data structure, which allows for quick access to all elements sharing the same key. Tree-based bags can thus be used to construct a nested hierarchy of bags, with each level representing a further partition of the full data set (in that case, smaller bags are tree-based bags themselves). In practice, tree bags are implemented as AVL trees mapping projections (keys) to sub-trees whose elements all project to the key under which the sub-tree is filed.

The search terms used to query tree-based bags are pairs, consisting of an optional key and a search term for sub-bags. The bfind operation behaves differently depending on the presence or absence of a key; if a key is given, then bfind returns the results of calling bfind with the additional search term on the smaller bag whose elements project to the given key. If no key is given, then bfind calls bfind on each smaller bag, and then merges all results (a process usually called a skip-scan, in the database world). Finally, benumerate is implemented by calling benumerate on each sub-tree and concatenating the resulting lists, and binsert is implemented by finding (or creating) the right sub-bag to insert into, and calling binsert on it. This design is similar to that found in most database management systems, where tuples are indexed on successive columns, with additional support for skip-scans. Figure 12 presents an example of such an indexed bag structure.

4.6 Automation

As an example of a very general optimization script, we implemented a tactic plan, which works automatically and is able to synthesize efficient implementations of a variety of query structures containing at most two tables, based on index structures backed by our bags library. Figure 13 shows an example of the code output by plan for our running Bookstore example. To produce this code from the reference implementation, the programmer only needs to specify a bag structure for each table, as shown in the part of Figure 1 after the method definitions. The plan tactic applies heuristics to rewrite each method into a more efficient form, given a set of available bag-based indexes. We will describe query heuristics in the most detail and then say a bit about refining mutators.

The query heuristics run to refine default (nondeterministic) method bodies induced by choices of abstraction relations. Here a relevant abstraction relation connects each table to a bag-based index. Default query bodies will compute with table contents specified as mathematical sets, and we need to rewrite those operations to use the indexes efficiently. Every actual code transformation is implemented

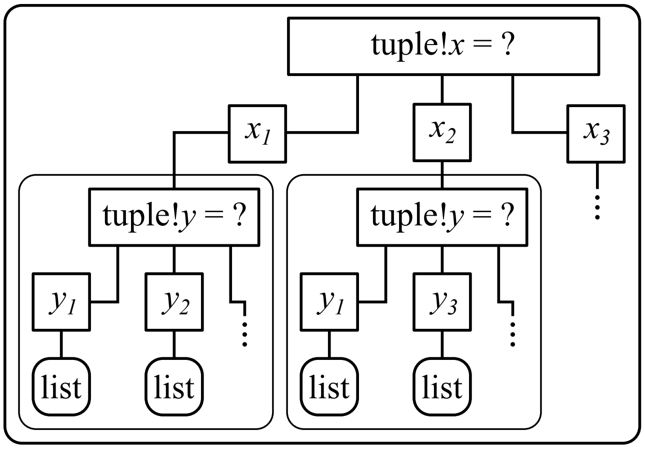


Figure 12. Indexed data is organized in a hierarchy of nested bags. In this example, the data set is first partitioned by column x, then by column y. Since our nested bags implementation supports skip-scanning, this same index can be used to answer queries related to x, to y, and to both x and y.

as a Coq setoid rewrite; we just need to determine a useful sequence of rules.

1. The **starting point** of refinement is expressions that work directly with mathematical sets. For example:

   $$\textbf{For}\ (r\ \textbf{in}\ T)\ \textbf{Where}\ (r.c_1 = 7)\ \textbf{Return}\ (r.c_2)$$

2. A **concretization step** replaces all references to sets with references to *lists* built by enumerating all members of index structures. For instance, the abstraction relation might declare that table T is implemented with index structure I, in which case we may rewrite the above to:

   $$\{\ell \mid \textbf{Permutation}\ \ell\ (\textbf{map}\ (\lambda r.\ r.c_2)\ (\textbf{filter}\ (\lambda r.\ r.c_1 == 7)\ (\textbf{benumerate}\ I)))\}$$

   That is, we convert set operations into standard functional-programming operations over lists, starting from the list of all table elements. Note that we use nondeterministic choice to select any permutation of the list that we compute. We do not want to commit to ordering this early, since we hope to find more efficient versions with different orders.

3. A **rewriting-modulo-permutation** step simplifies the list expression, possibly with rules that change ordering. Here we use standard algebraic laws, like $\textbf{map}\ f\ (\textbf{map}\ g\ \ell) = \textbf{map}\ (f \circ g)\ \ell$. Less standard rules locate opportunities to apply our index structures. Our example query would be refined as follows, assuming the index structure only covers table column c_1:

   $$\{\ell \mid \textbf{Permutation}\ \ell\ (\textbf{map}\ (\lambda r.\ r.c_2)\ (\textbf{bfind}\ I\ (7, [])))\}$$

   Our tactic analyzes **filter** functions syntactically to figure out useful ways of applying index structures. In full generality, the heuristics of this phase apply to filter conditions over two tables, and they are able to decompose conjunctive conditions into some use of indexes and some use of less efficient filtering for conditions that do not map neatly to the indexes.

4. A **commitment** step accepts the current list expression as the final answer, commiting to an ordering. Our example is refined in one simple step to:

   $$\textbf{ret}\ (\textbf{map}\ (\lambda r.\ r.c_2)\ (\textbf{bfind}\ I\ (7, [])))$$

```
Sharpened ADTRep (TBookStorage ∗ TOrderStorage) {
  const Init (_: ()) : rep :=
    ret (bempty, bempty)

  meth PlaceOrder (r_n: rep , o: Order) : bool :=
    let (books, orders) := r_n in
    if bcount books (None, Some o!ISBN, []) ≠ 0
    then ret (books, binsert orders o, true)
    else ret (books, orders, false)

  meth AddBook (r_n: rep , b: Book) : bool :=
    let (books, orders) := r_n in
    if bcount books (None, Some b!ISBN, [λ b'.
                         b!Title ≠ b'!Title ||
                         b!Author ≠ b'!Author]) == 0
    then ret (binsert books b, orders, true)
    else ret (books, orders, false)

  meth GetTitles (p: rep , author: string) : list string :=
    let (books, orders) := p in
    ret (books, orders,
          map (λ tuple. tuple!Title)
                (bfind books (Some author, None, [])))

  meth NumOrders (p: rep , author: string) : nat :=
    let (books, orders) := p in
    ret (books, orders,
         fold_left
           (λ count tup.
              count + bcount orders (Some tup!ISBN, []))
           (bfind books (Some author, None, [])) 0)
}
```

Figure 13. Synthesized code for Bookstore example

This basic three-step process can be extended quite flexibly. Our implementation handles aggregate functions (e.g., count or max) in the same way. A use of such an operation is **concretized** to a fold over a list, and we apply **rewriting** to incorporate chances to use index structures to compute folds more directly.

One further subtlety applies in the rewriting step for queries over multiple tables. Concretization rewrites a join of two tables into a Cartesian-product operation **Join_Lists**, defined as follows, where **flat_map** is a variant of **map** that concatenates together the lists produced by its function argument.

$$\textbf{Join\_Lists}\ \ell_1\ \ell_2 \equiv \textbf{flat\_map}\ (\lambda a.\ \textbf{map}\ (\lambda b.\ (a, b))\ \ell_2)\ \ell_1$$

This code pattern is known as a nested loop join, in the database world. Notice that it is inherently asymmetric: we handle one table in an "outer loop" and the other in an "inner loop." Imagine that, because of a **Where** clause, our nested loop has been concretized within a **filter** call, like so:

$$\textbf{filter}\ (\lambda(a, b).\ a.c_1 = 7)\ (\textbf{Join\_Lists}\ \ell_1\ \ell_2)$$

Note that this filter condition is highly selective when it comes to rows of ℓ_1, but it accepts any row of ℓ_2. Rewriting will apply the following algebraic law to push the **filter** inside the **Join_Lists**:

$$\begin{array}{ll} & \textbf{filter}\ f\ (\textbf{Join\_Lists}\ \ell_1\ \ell_2) \\ = & \textbf{flat\_map}\ (\lambda a.\ \textbf{map}\ (\lambda b.\ (a, b))\ (\textbf{filter}\ (\lambda b.\ f\ (a, b))\ \ell_2))\ \ell_1 \end{array}$$

Here we see that the inner **filter** can be refined into an efficient use of **bfind**, *if* we first swap the order of the **Join_Lists** arguments. The plan tactic attempts heuristically to orchestrate this style of strategic rewriting.

The heuristics for queries are the heart of what plan does, but it also optimizes insert operations. Subsection 4.3 explained how we refine constraint checks into queries, to reuse query-based refinement tools. A plan invocation is responsible for noticing

```
ADT BookstoreCache implementing BookstoreSig {
  rep := UnConstrQS BookstoreSchema × (CacheSpec 𝕊 ℕ).rep
  const Init (_: unit) : rep := (empty, ∅),

  meth AddBook (r : rep, book: Book) : rep :=
    (r_q, r_c) ← Insert book into Books;
    c ← Count (For (b in Books)
               Where (author = b!Author)
               Return ());
    ret (r_q, CacheSpec.update (r_c, book!Author, λ_.c)),

  meth NumTitles (r : rep, author : string) : nat :=
    let (r_q, r_c) := r in
      n ← CacheSpec.lookup(r_c, author);
      If n = Some n' Then ret n'
      Else c ← Count (For (b in Books)
                      Where (author = b!Author)
                      Return ());
       ret (r_q, CacheSpec.insert (r_c, c) , c),

   query GetTitles (author: string) : list string := ... }
```

Figure 14. Version of the bookstore ADT that caches queries

opportunities to apply a suite of formal checks-to-queries rules. We also apply a set of refinements that remove trivially true checks and prune duplicate checks. After the set of checks has been simplified as much as possible, we do a case analysis on all the ways that all the checks could turn out, performing an actual table insert only in cases whose checks imply validity.

4.7 Caching Queries

Expressing refinement using the small set of core ideas presented so far allows us to soundly and cleanly integrate optimizations from different domains. As a demonstration, this section will show how we integrate the cache ADTs developed in Section 3 to cache query results. Importantly, we will apply this refinement after dropping the data-integrity constraints, but *before* implementing the query structure. Moreover, we perform this refinement using the CacheSpec reference implementation, allowing users to choose any caching implementation independently of the query structure. We will be refining a variation of the bookstore ADT that replaces the GetTitles method with one that counts the number of books an author has written:

```
query NumTitles (author: string) : list string :=
  Count (For (b in Books)
         Where (author = b!Author)
         Return ())
```

We begin by honing the representation of the reference query structure to include a cache ADT keyed on author names. The abstraction relation used to hone the representation maintains the invariant that the cache of the new representation only holds valid book counts for each author:

$$\begin{array}{rl} r_v \approx_c (r'_v, r'_i) & \equiv r'_v = r_v \\ & \wedge\ \forall a, v.\ (a, v) \in r'_i \rightarrow \mathsf{NumTitles}(r_v, k) \rightsquigarrow v \end{array}$$

We can use the cache consistency predicate to refine NumTitles so that it first nondeterministically picks a key in the cache using lookup, returns that value if it exists, and otherwise runs the query and adds a new key to the cache (this updated cache trivially satisfies the representation invariant). NumOrders and PlaceOrder are unchanged. Since AddBook can increase the number of books for an author in the cache, it needs to update the cache if the author of the new book is in the cache. Applying all these refinements to the bookstore ADT from Figure 11 yields the refined ADT in Figure 14.

Observe that the following refinement holds:

$$\mathbf{For}\ (x\ \mathbf{in}\ R \cup \{a\})\ b \quad \supseteq \quad \begin{array}{l} l \leftarrow b\ a; \\ l' \leftarrow \mathbf{For}\ (x\ \mathbf{in}\ R)\ b; \\ \mathsf{Return}\ (\mathsf{l} +\!\!+ \mathsf{l'}) \end{array} \tag{2}$$

thus allowing us to refine the query in AddBook:

```
Count( l ← Where (author = a!Author) Return ();
       l' ← For (b in Books)
              Where (author = b!Author) Return ();
       Return (l ++ l'))
```

which can be further refined into:

```
n ← Count (Where (author = a!Author)
           Return ());
n' ← Count (For (b in Books)
            Where (author = b!Author)
            Return ());
Return (n + n')
```

Due to the representation invariant in the abstraction relation, we know that the query bound to n' returns precisely the cache contents, allowing us to refine the update to simply increment the value of author currently in the cache:

```
meth AddBook (r : rep, book: Books) : rep :=
  (r_q, r_c) ← Insert book into Books;
  ret (r_q, CacheSpec.update (r_c, book!Author, increment))
```

This case is actually an example of the broader *finite differencing* [13] refinement for improving the performance of an expensive operation f whose input can be partitioned into two pieces, old y and new x, such that $f(x \oplus y) = f'(x) \oplus' f(y)$. The representation invariant ensures that each value in the cache already stores $f(y)$, allowing us to reduce the computation of $f(x \oplus y)$ to computing $f'(x)$ and updating the value in the cache appropriately. Note that Fiat's refinement process supports finite differencing of *any* ADT operation, through the use of abstraction relations to express cache invariants and refinements to replace repeated computations. In particular, (2) allows us to easily cache **For** queries by partitioning their results into novel results and the portion already in the cache when performing **Insert** updates.

5. Evaluation

5.1 Extraction of the Bookstore Example

Once they have been fully refined, our data structures can be extracted to produce verified OCaml database management libraries. We performed such an extraction for the bookstore example and benchmarked it; the observed performance is reasonable, and the operations scale in a way that is consistent with the indexing scheme used, as demonstrated by Figure 15. For instance, starting with an empty database, it takes about 480 ms on an Intel Core i5-3570 CPU @ 3.40GHz to add 10 000 randomly generated books filed under 1 000 distinct authors, and then 6.8 s to place 100 000 orders. Afterward, the system is able to answer about 350 000 GetTitles queries per second, and about 160 000 NumOrders queries per second.

5.2 Further Examples

To verify the generality of our automated refinement strategies, we studied two more examples: a weather-data database and a stock-market database, both of which are included in the accompanying archive.

The weather example includes two tables – one to hold information about the weather stations, and one to log the measurements – and supports operations such as counting the number of stations in a given geographic area, or computing the highest temperature

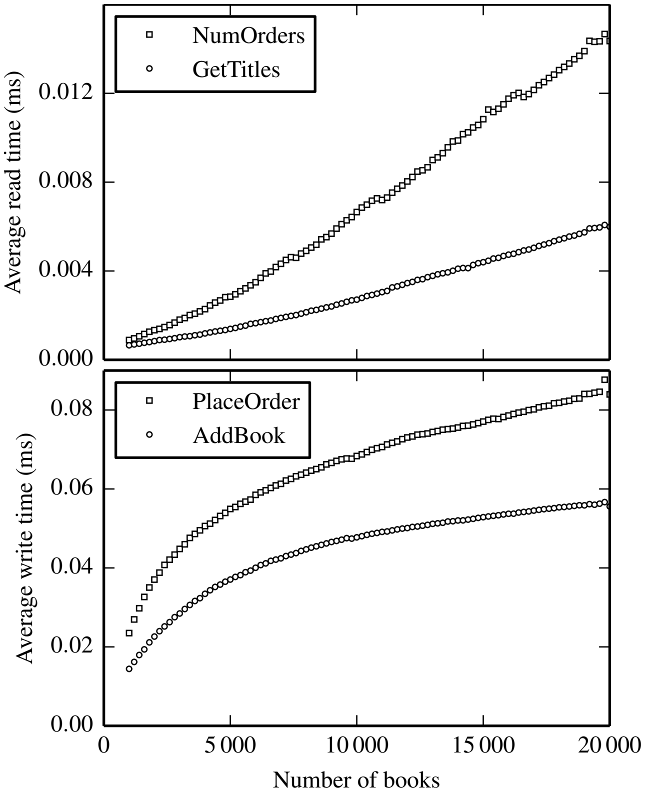


Figure 15. Average query execution time, for increasingly large bookstores

on one day. The stock market example includes one table listing information about stocks and one table keeping track of transactions, and allows clients to compute the total volume of shares exchanged on one day for a particular stock, plus the largest transaction for a given type of stock.

In both cases skip-scanning allowed for space-efficient indexing (in the weather case, stations could produce a small number of different measurement types: wind speed, temperature, air pressure, or humidity; in the stocks case, different types of stocks were distinguished), and in both cases non-trivial functional dependencies were expressed (for examples, two transactions occurring at the same time and concerning the same stock could differ in the number of shares exchanged, but not in price). The `plan` tactic synthesizes correct, efficient implementation code in both cases.

6. Related Work

The concept of deriving implementations that are correct by construction via stepwise refinement of specifications has been around since at least the late sixties [7].

Deductive Synthesis Specware[20] and its predecessors KIDS [16] and DTRE [2] are deductive synthesis tools for deriving correct-by-construction implementations of high-level specifications. Specware is accompanied by a library of domain theories that describe how to do iterative decomposition of high-level specifications into subproblems until an implementation can be constructed. At each step, Specware checks the validity of the refinement, although only recently have they begun generating Isabelle/HOL proof obligations justifying these transformations. At the end of refinement, Specware has a series of automated (and, as we understand them, quite sophisticated) transformations that generate C code, though these final steps are currently unverified. In contrast, derivations of ADTs with Fiat are completely verified by Coq, and these derivations may be integrated within larger, more general proof developments. We have also demonstrated more automated refinement for more restrictive domains, as in our query structure examples.

Synthesis of Abstract Data Types There is also a philosophical difference between Fiat and Specware – Kestrel has focused on using Specware to develop families of complex algorithms, including families of garbage collectors [14], SAT solvers [17], and network protocols. In contrast, we envision Fiat being used to generate high-assurance code for algorithmically "simple" domains that are amenable to automation. A number of domains have been shown to have this property: Paige et. al [12] demonstrate how efficient implementations of ADTs supporting set-theoretic operations can be derived by applying fixed-point iteration [12] to generate initial implementations, using finite differencing [13] to further optimize the resulting implementation, before finally selecting data structure implementations. The generation of data types supporting query-like operations is another such domain. P2 [1] was a DSL extension to C that allowed users to specify the layout of container data structures using a library of structures implementing a common interface, akin to Fiat's Bag interface. This interface included an iterator method for querying contents of the container – implementation of these iterators would dynamically optimize queries at runtime. More recently, Hawkins et. al [8] have shown how to synthesize the implementation of abstract data types specified by abstract relational descriptions supporting query and update operations. They also provide an autotuner for selecting the best data representation implementation in the space of possible decompositions. Our work with Fiat expands on these past projects by adding proofs of correctness in a general-purpose proof assistant, which also opens the door to sound extension of the system by programmers, since Coq will check any new refinement rules.

Constraint-Based Synthesis Constraint-based synthesis formulates the synthesis problem as a search over a space of candidate solutions, with programmers providing a set of constraints to help prune the search space. The Sketch [18] synthesis system, for example, allows programmers to constrain the search space by encoding their algorithmic insight into skeleton programs with "holes" that a synthesizer automatically completes. Sketching-based approaches have been used to synthesize low-level data-structure manipulating algorithms [15], concurrent data structures [19], and programs with numeric parameters that are optimized over some quantitative metric [4]. These approaches fit into the broad decomposition of "functionality + optimization" proposed here, with the initial sketch representing the former and the synthesizer providing the optimization. Excitingly, Fiat enables opportunities for programmers to inject further insight into the synthesis process by chaining together honing tactics with various degrees of automation. Subsection 4.7 provides an example of such an development, with the user first automatically dropping data-integrity constraints via a honing tactic before manually specifying how to cache certain queries.

Reasoning with Refinements Cohen et el. have verified an algebra library in Coq using data type refinement [5] by verifying algorithms parametrized over the data type and its operations. Verification is done using a simple, "proof-oriented" data type. The authors then show how to transport the proof of correctness to a version of the algorithm using a more efficient implementation that is related to the proof-oriented data type by a refinement relation, akin to the abstraction relations used in Fiat.

7. Future Work and Conclusion

We have reported here on the start of a project to explore the use of proof assistants to enable a new modularity pattern in programming: separating *functionality* from *performance*, where programmers express their functionality and then apply *optimization scripts* to refine it to efficient implementations. Special-case systems like SQL engines have provided this style of decomposition, but only for hardcoded domains of functionality. We explained how the design of Fiat allows programmers to define new functionality domains and new optimization techniques, relying on Coq to check that no optimization technique breaks program semantics. Programmers think of implementing a new database engine as a big investment, but they think of implementing a new container data structure as a reasonable component of a new project. The promise of the Fiat approach is to make the former as routine as the latter, by giving that style of optimization strategy more first-class status within a programming environment, with *enforced modularity* via checking of optimization scripts by a general-purpose proof kernel.

We plan to explore further applications of Fiat, both by identifying other broad functionality domains that admit effective libraries of optimization scripts, and by narrowing the gap with hand-tuned program code by generating low-level imperative code instead of functional code, using the same framework to justify optimizations that can only be expressed at the lower abstraction level.

References

[1] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 1993.

[2] Lee Blaine and Allen Goldberg. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, pages 165–204. Elsevier, 1991.

[3] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1), March 1994.

[4] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2014.

[5] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs*. Springer International Publishing, 2013.

[6] http://coq.inria.fr/distrib/current/refman/Reference-Manual029.html.

[7] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. Circulated privately, August 1967.

[8] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011.

[9] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer Berlin Heidelberg, 1986.

[10] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[11] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Symposium on Very high level languages*, New York, NY, USA, 1974. ACM.

[12] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code – a case study. *J. Symb. Comput.*, 4(2):207–232, October 1987.

[13] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3), July 1982.

[14] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Formal derivation of concurrent garbage collectors. In *Mathematics of Program Construction*, pages 353–376. Springer Berlin Heidelberg, 2010.

[15] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011.

[16] Douglas R. Smith. KIDS: A semi-automatic program development system. In *Client Resources on the Internet, IEEE Multimedia Systems '99*, pages 302–307, 1990.

[17] Douglas R. Smith and Stephen J. Westfold. Synthesis of propositional satisfiability solvers, 2008.

[18] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[19] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.

[20] http://www.kestrel.edu/home/prototypes/specware.html.

[21] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.