Extensible Data-Representation Selection for Correct-by-Construction Program Derivation

Abstract

Program synthesis via refinement is a venerable approach for gradually transforming specifications into executable code, generating a proof trail showing that the final efficient program adheres to the specification. We present the first automated, proof-generating refinement system that invents new data structures to suit the needs of a program, applying global program analysis to understand those needs. Our system, based on the Fiat library for the Coq proof assistant, is extensible not just with new data structures but also with new rules for inventing more complex data structures by combining simpler ones. All data-structure operations come with semantic interfaces capturing their behavior in terms of abstract mathematical objects such as sets, and the system soundly and automatically tiles the specification with those operations to generate an OCaml-style program and a proof of its correctness. Starting from a high-level specification in extensible SQL-like language using logic to describe arbitrary operations on data, without explaining how to execute them, our system automatically matches those operations to proof-generating data-structure-construction rules in our library to build an efficient, correct-by-construction implementation.

1. Introduction

Whether objects, abstract data types (ADTs), or modules, every modern programming language features some *data abstraction* mechanism for hiding internal details of data implementations behind an abstraction boundary. The benefits to programmers are two-fold: by implementing their code against an interface, clients of an abstraction can safely link in any valid implementation, enabling them to explore a range of implementations to select the best fit. Similarly, authors of an abstraction have complete flexibility when choosing the implementation of encapsulated state, freed from the worry of breaking client code. In both cases, programmers rely on the language enforcing the abstraction boundary to guarantee the correctness of the final program.

These two benefits are key to the *data-representation synthesis* problem [5], wherein programmers give a high-level specification of an abstract data type and rely on a synthesizer to find an optimal data representation and method implementations. In this setting, the abstraction boundary enables the synthesizer to freely select a concrete data representation. Instead of programming against a rigid interface and writing operations like "look up according to the value of this key," clients can code against very general logical specifications, much like a client of relational databases (e.g., SQL). Extensibility poses an important challenge in this setting: users should be able to extend the set of possibile data representations considered by the synthesizer in a way that does not compromise any assurances about the correctness of the produced code. This paper presents a novel solution to this challenge, allowing a synthesizer to be augmented with a package containing new data-representation strategies and specifications of their behavior, with the underlying system guaranteeing the correctness of any resulting implementation. We have incorporated our solution into the Fiat framework [4] for deductive synthesis of ADTs.

To make our discussion more concrete, consider the specification in Figure 1a of an abstract data type that an OS implementor might use to store information about running processes. This specification models its internal state as a multiset or *bag* of records ProcTbl; the SQL-like notation used to specify its methods reduces to basic operations of set theory on this bag. In contrast to standard SQL, we allow specifications to appeal to user-defined mathematical predicates (coded in Coq) as filters in **Where** clauses. Figure 1b presents a *refined* version of Figure 1a, in the sense that any behavior allowed on the right is also allowed on the left. This ADT is similar to what a functional programmer might write, although the code uses features not usually included in executable programming languages, i.e. general set comprehensions and union operations. This ADT is mostly complete, in that we can produce an executable implementation merely by selecting a concrete data representation for the bag and matching any corresponding bag operations with the methods of the representation we chose. There is an "obvious" näive implementation of this spec, representing bags as unsorted lists, but that representation would bring unacceptable performance for large data sets. We would prefer for the compiler to find a more efficient representation for us automatically, by analyzing which sorts of operations the spec performs on the data. In our process scheduler, for example, the final data structure should efficiently support the two specialized set comprehensions that appear in Figure 1b, {x | x!state= state} and {x | start $\leq$ x!time$\leq$ end}.

This paper presents a system that automatically implements partial ADT implementations like that in Figure 1b by selecting concrete data structures that *efficiently* support the necessary abstract operations, and *automatically* deriving implementations of the ADT's methods in terms of its selection.

```
ADT {
  Rep ProcTbl :=
    MultiSet of ⟨ pid :: ℕ, state :: State, time :: ℕ⟩
                  enforcing UNIQUE pid,

  Constructor Init : rep := ret ∅ ,
  Method Spawn (r : rep) (time : ℕ) : rep × bool :=
    new_pid ← { pid' | ∀ proc, proc ∈ (r!ProcTbl)
                              → (pid' ≠ proc!pid)};
    Insert ⟨ pid::new_pid, state::ASLEEP, time::time⟩
           into r!ProcTbl,
  Method Enum (r : rep)(state : State) : rep × list ℕ:=
    For (p in r!ProcTbl)
    Where (p!state = state)
    Return (p!pid),
  Method ByTime (r : rep)(start end : ℕ) : rep× list ℕ:=
    For (p in r!ProcTbl)
    Where (start ≤ p!time ≤ end)
    Return (p!pid) }
```

(a)

≻≈

```
ADT {
  Rep ProcTbl :=
    MultiSet of ⟨ pid :: ℕ, state :: State, time :: ℕ⟩ ,
  Rep max := ℕ,

  Constructor Init : rep := ret (∅ , 0),
  Method Spawn (r : rep) (time : ℕ) : rep × bool :=
    ret (r!ProcTbl ∪ {⟨ pid::max, state::ASLEEP, time::time⟩ },
         max + 1, true)
  Method Enum (r : rep) (state : State) : rep × list ℕ:=
    ps ← {ps | ps ≃ { x ∈ r!ProcTbl | x!state = state }}
    ret (r, map (λp. p!pid) ps),
  Method ByTime (r : rep) (start end : ℕ) : rep × list ℕ:=
    ps ← {ps | ps ≃ { x ∈ r!ProcTbl | start ≤ x!time ≤ end} }
    ret (r, map (λ p. p!pid) ps) }
```

(b)

Figure 1. Fiat specification for a process-scheduling database

Our approach can be decomposed into the following major steps:

- We first examine the body of each method to identify every abstract operation on bags that could be efficiently implemented by picking the right data structure, building a "wishlist" of such operations. We then heuristically choose the operations that the eventual data representation will support.
- Next, we automatically implement each high-level operation in terms of the selected operations, demonstrating along the way that this choice is sufficient to derive a complete, executable implementation.
- We construct the final implementation by building concrete data structures implementing the selected data-representation operations.

Each step of this approach can be extended with new implementation hints, allowing users to incorporate new data structures with a level of effort on par with implementing and linking in a (verified) implementation of a standard interface. In fact, we go beyond merely choosing from a library of verified data structures. We also allow extensions to suggest *new recipes for composing data structures*, producing hybrid structures that combine the performance advantages of their constituents. We implement this approach as the final (automated and proof-generating) stage in the synthesis pipeline, outlined in Figure 3, for compiling an ADT specification like Figure 1a into an executable, correct-by-construction functional program. Figure 2 shows the nested collection of range trees, AVL trees, and lists automatically selected by this pipeline for the process scheduler example.

The question of efficient data-structure selection has seen considerable attention over the years in both the programming languages and formal methods communities. Systems

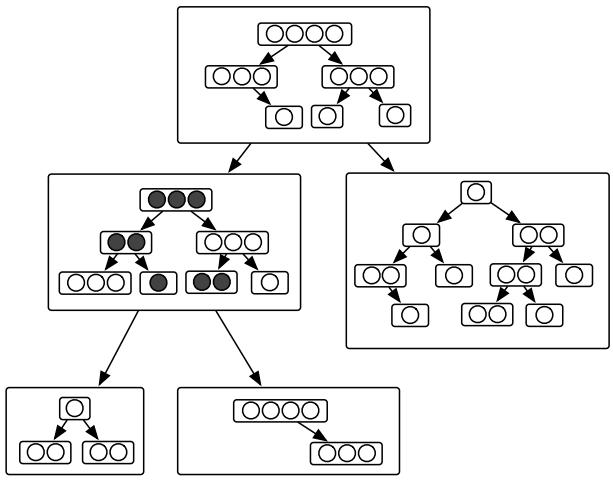


Figure 2. A data structure supporting compound searches by nesting a range tree within an AVL tree. The gray internal nodes represent the subnodes touched by a range query; using naive AVL trees for both levels would visit all the subnodes.

like CHAMELEON [19] and COCO [25] *dynamically* monitor run-time usage of implementations of the fixed collections interface to either provide users with suggestions for better implementations in the case of CHAMELEON or to automatically switch implementations at run-time in the case of COCO. In contrast, our goal here is to *statically* synthesize the best data structure for specifications written in terms of an abstract model of state instead of a collections interface. The essence of our approach could be implemented in static metaprogramming approaches for languages such as Scala and Racket [18, 23], although we provide *stronger* guarantees about the correctness of the implementation we produce. In contrast to other automatic, proof-generating data refinement

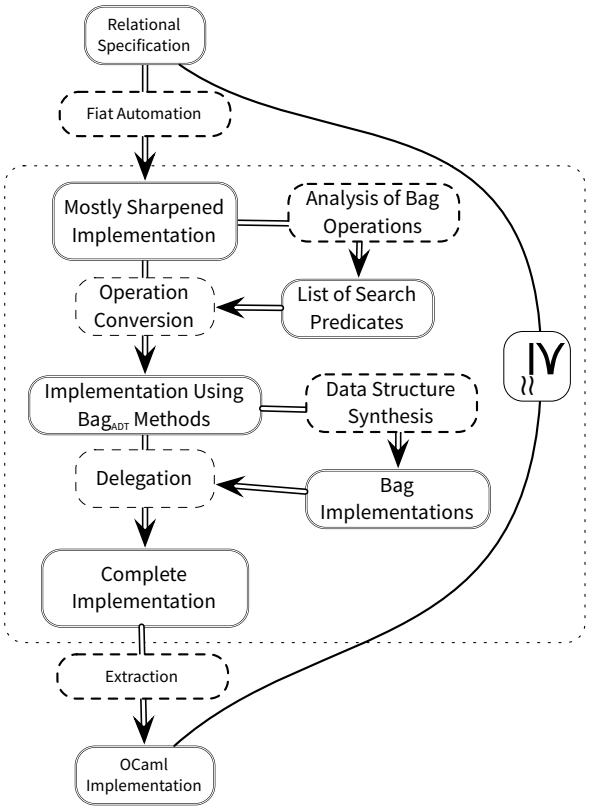


Figure 3. Anatomy of data-structure selection.

frameworks [3, 10], our solution differs in first, considering *every* operation on abstract state in order to select an efficient data representation, and second, in automatically constructing *on-demand data types* based on that analysis.

We begin by reviewing the foundations of deductive refinement underlying the Fiat system before fleshing out each of the high-level steps in Figure 3. At each step, we detail the extensions needed to incorporate new data structures and novel implementation strategies into the pipeline. We finish this discussion with an overview of how these strategies are packaged and deployed within the Fiat framework, before demonstrating the applicability of our system with a series of case studies that synthesize a number of complicated nested data structures. The code for both the extended Fiat system and the case studies is compatible with the latest release of Coq (8.4pl5) and is included in the accompanying anonymous supplement.

2. An Overview of Fiat

We begin with a deeper explanation of the example from Figure 1 in order to illustrate the foundations of the Fiat framework's approach to deductive synthesis, which underlies our approach to data-representation synthesis. The starting point for a Fiat derivation is an abstract data type [13], which encapsulates optimizable state in its representation type. As noted in the introduction, the internal state of the specification in Figure 1a is modelled as a bag of records with fields for the ID, current state, and running time of a process (pid, state, and time, respectively). This model is also equipped with the *representation invariant* that each record has a unique ID. The operations are specified in terms of this abstract model of state; each operation specifies a set of values using the nondeterminism monad, where each value in this set represents a possible output in any eventual implementation of this specification. The Spawn method, for example, uses this monad's set-comprehension combinator to defer selection of a fresh ID to the implementation:

$\{\text{pid}' \mid \forall\ \text{proc},\ \text{proc} \in (\text{r!ProcTbl}) \rightarrow (\text{pid}' \neq \text{proc!pid})\}$

Each operation is specified in Gallina, Coq's functional specification and programming language. The SQL-like notation used in Figure 1a is defined in Fiat's Query Structure library using Coq's extensible parser and desugars into Gallina according to the semantics outlined in the original Fiat paper. We restrict our data representations to a single "table" for clarity of presentation, but the library imposes no such restrictions on the number of tables used in a specification. In contrast to standard SQL, the **Where** clauses use arbitrary mathematical predicates as filters. As a consequence, the Query Structures library out of the box supports writing filtering conditions based on a variety of natural mathematical notions that are not hardcoded into the library, for example prefix-matching conditions on lists. Fiat also supports arbitrary representation invariants, allowing users of the Query Structures library to specify data-integrity constraints that go beyond those of standard SQL.

This specification is iteratively refined using a set of automated honing tactics, eventually arriving at the partial implementation in Figure 1b. The ADT's data representation has been augmented with a cache of the greatest ID, which Spawn uses to select new process IDs. Each transformation is accompanied by a proof, certified by Coq, that the resulting ADT is a refinement of the initial specification, written $\texttt{adt}_1 \succeq_{\approx} \texttt{adt}_2$, under an abstraction relation [6] $\approx$ between the internal states of the two ADTs. This proof ensures that the operations of $\texttt{adt}_2$ will take states related by $\approx$ to states related by $\approx$ while producing no outputs that $\texttt{adt}_1$ cannot also produce, allowing an ADT client to reason about an implementation's behavior solely through the original specification. This refinement proof is what allows the representation invariant to be omitted from the partial implementation, as it follows from the abstraction relation used to derive the implementation. The Query Structures library provides honing tactics that automate refinements of specifications written in its SQL-like language. These tactics automatically translate **Where** clauses into set-comprehension operations on the appropriate relation and insert dynamic checks that preserve data-integrity constraints.

The result of this refinement process is the partial ADT implementation in Figure 1b. We can now formally define these

```
Variables ItemT searchT Match updateT AppUpdate.
Definition Bag_ADT :=
 ADTRep (MultiSet ItemT) {
  Def Constructor Empty : rep := ret EMPTY,
  Def Method Enumerate (r : rep) : rep × list ItemT :=
    l ← {l | l ≃ r}; ret (r, l),
  Def Method Find (r : rep) (st : searchT) : rep × list ItemT :=
    l ← {l | l ≃ {x ∈ r | Match st x = true}}; ret (r, l),
  Def Method Count (r : rep) : rep × nat :=
    l ← {l | l ≃ r }; ret (r, |l| ),
  Def Method Insert (r : rep) (item : ItemT) : rep :=
    ret (r ∪ {item}),
  Def Method Delete (r : rep) (st : searchT)
      : rep × list ItemT :=
    l ← {l | l ≃ r };
    ret ({x ∈ r | Match st x = false}, filter (Match st) l),
  Def Method Update (r : rep) (st : searchT) (ut : updateT)
      : rep × list ItemT :=
    l ← {l | l ≃ {x ∈ r | Match st x = true }};
    ret ({x ∈ r | Match st x = false}
         ∪ {x ∈ map (AppUpdate ut) l}, l)}.
```

Figure 4. Fiat specification of the Bag ADT

partial implementations as those where the mathematical model of state is the *only* source of nondeterminism. We also refer to such partial implementations as *mostly sharpened* ADTs. Any choice of a concrete data structure supporting the operations on this model is enough to build a complete ADT implementation. These sorts of mostly sharpened ADTs form the starting point for our approach to extensible and adaptive data-representation synthesis.

3. Data-Representation Selection

In order to select an appropriate data representation for a mostly sharpened ADT, we must first precisely characterize the operations that it needs to support. In our process-scheduler example, these operations are the union operation used to insert a tuple into the database and the set comprehension operations used in the Enum and ByTime methods. Importantly, instead of a single generic set-comprehension operation, the final implementation of ProcTable need only support the specialized searches represented by { x ∈ r!ProcTbl | x!state = state } and { x ∈ r!ProcTbl | start ≤ x!time≤ end }.

The Fiat ADT $\mathsf{Bag}_{\mathsf{ADT}}$ in Figure 4 captures all the operations a bag implementation should support. Following the Fiat style, the code in the figure gives a nondeterministic reference implementation of bags, leaving plenty of leeway for implementers to choose representations and optimizations. In contrast to standard interfaces that only give method signatures, this specification captures the *behaviors* of the methods, as enforced by the ADT refinement relation, forming a *semantic interface* for the ADT. Any data structure that satisfies the $\mathsf{Bag}_{\mathsf{ADT}}$ semantic interface can be added to the toolbox of available implementation strategies. Moreover, it can be added as a true library, without making changes to code written previously for the core framework or for other data structures.

$$\frac{P \sqsupseteq \ell_P \quad b \sqsupseteq \ell_b}{\{x \in r | Px\}\ b \sqsupseteq \ell_P \cup \ell_b}\ \textsc{IDWhere} \qquad \frac{P \sqsupseteq \ell_P \quad Q \sqsupseteq \ell_Q}{P \wedge Q \sqsupseteq \ell_P \cup \ell_b}\ \textsc{IDAnd}$$

$$\frac{}{X \sqsupseteq [\,]}\ \textsc{IDDefault}$$

$$\frac{\overline{b_i \sqsupseteq \ell_i} \qquad \overline{b_j \sqsupseteq \ell_j}}{\begin{array}{l}\mathsf{QueryStructure}\ X\ \{ \\ \quad \overline{\mathsf{Def\ Constructor}\ c_i\ \overline{p_i} : \mathbf{rep} := b_i}, \\ \quad \overline{\mathsf{Def\ Method}\ m_j\ \overline{p_j} : \mathbf{rep} := b_j}\}\end{array} \sqsupseteq \bigcup_i \ell_i \cup \bigcup_j \ell_j}\ \textsc{IDADT}$$

Figure 5. Example syntax-tree exploration rules.

An important feature of the $\mathsf{Bag}_{\mathsf{ADT}}$ specification is that its Find method captures specialized set comprehension via two parameters: searchT, representing a type of search terms, and Match, which captures the search semantics:

∀ (r : **rep**) (st : searchT). Find r st ≃ {x ∈ r | Match st x = true}

In other words, Find r st returns a list (in some order) of all of the elements of r that Match the search term st. The specifications for other data-structure operations (e.g. Insert, Delete) are similarly expressed in terms of these parameters. Figure 12 includes the value of Match for AVL trees, which has the natural definition that a record t with field f matches a key k iff f has value k.

3.1 Automatic Data-Representation Selection

The first step in automatically implementing a partial ADT is to identify all occurrences of the operations on the abstract state, so that they can *all* be taken into account when choosing an appropriate set of semantic interfaces. In the case of ADTs specified using the Query Structures library, this is done by analyzing each method of the ADT specification to identify search clauses that can be efficiently implemented with appropriate selections of the searchT and Match parameters. Conceptually, this analysis is implemented as a backtracking search using a set of declarative rules mapping clauses to descriptions of the operations that would be helpful in implementing them. The judgments of these rules look like the following: $[\mathsf{ADT}\ \{\ \ldots\ \} \sqsupseteq [\mathsf{EQ}_f; \leq_g; \ldots]]$ This rule concludes that an equality-based index on field f and an ordering-based index on field g, among others, may be useful, because the ADT specification applies $=$ to f values and $\leq$ to g values. We want our formalism to support extension with new specification patterns and any new sorts of summaries that ought to be recorded when we see those patterns.

Intuitively, the deduction rules can be divided into a generic set exploring the syntax tree of the ADT, as in Figure 5, and a set of rules specific to patterns of set-theory code, as with the rules in Figure 6a for recording attribute uses inside of equality predicates. Extending this search

$$\frac{}{x!f = v \sqsupseteq [\mathsf{EQ}_f]}\ \mathrm{ID}_{EQ}\mathrm{L} \qquad \frac{}{v = x!f \sqsupseteq [\mathsf{EQ}_f]}\ \mathrm{ID}_{EQ}\mathrm{R}$$

$$\frac{}{x!f = y!g \sqsupseteq [\mathsf{EQ}_f;\ \mathsf{EQ}_g]}\ \mathrm{ID}_{EQ}\mathrm{ALL}$$

(a) $\mathsf{Match}_{\mathsf{EQ}}$

$$\frac{}{x!f \le v \sqsupseteq [\le_f]}\ \mathrm{ID}_{\le}\mathrm{L} \qquad \frac{}{v \le x!f \sqsupseteq [\le_f]}\ \mathrm{ID}_{\le}\mathrm{R}$$

$$\frac{}{x!f \le y!g \sqsupseteq [\le_f;\ \le_g]}\ \mathrm{ID}_{\le}\textsc{Both}$$

$$\frac{}{x!f \le y!g \le v \sqsupseteq [\le_f;\ \le_g]}\ \mathrm{ID}_{\le}\textsc{All}$$

$$\frac{}{v \le x!f \le y!g \sqsupseteq [\le_f;\ \le_g]}\ \mathrm{ID}_{\le}\textsc{AllR}$$

(b) $\mathsf{Match}_{\le}$

Figure 6. Identifying candidate attributes for index optimization.

to associate new search predicates with new indexes is accomplished with the addition of rules. Figure 6b contains an example of the rules that a programmer might add to support range queries.

The list of attribute uses for the ProcTable relation in the process scheduler, for example, is $[\mathsf{EQ}_{\mathsf{state}}; \le_{\mathsf{time}}]$. Once this list has been gathered, we heuristically choose which operations we should focus on supporting efficiently. For instance, we can give higher priority to those operations that appear more frequently. The choice of operations determines which concrete data structures later synthesis phases will consider. The choice phase associates each bag in the specification with a *logical index* ℓ that instantiates the parameters on the first line of Figure 4, fixing what type of *search term* the data structure supports directly, along with the semantics that should be applied to those search terms.

4. Implementing Multiset Operations

Having characterized the operations on an eventual data representation via a semantic interface, it remains to select concrete data structures and to implement the methods of the ADT in terms of this selection. The use of semantic interfaces cleanly decomposes this into two subtasks: (1) the implementation of the methods of the original ADT in terms of the data representation's interface and (2) the selection of concrete data structures implementing that interface. Using the same abstract model of state in both the original ADT and the derived semantic interfaces allows for a clean transition between operations on abstract state and calls to the methods of the semantic interface. The key challenge is mapping the environment of the former into parameters for the latter.

As a concrete example, consider the case of implementing the two query methods from Figure 1 for an ADT representing the ProcTable relation with an AVL tree mapping State values to lists of tuples. For this data structure, a search term is a pair of an optional State value (giving the required value of the state column if present) and a Boolean predicate over messages (giving any other filter conditions). The corresponding match function compares the first component of the search term to the state attribute of a tuple and applies the second component to the entire tuple:

```
{ searchT := nat_opt × (Tuple → bool);
  Match := λst, r. Match_EQState st_π1 r ∧ Match_⊥ st_π2 r }
```

The body of ByTime already closely mirrors the specification of the Find method from Figure 4, a fact captured by the following *refinement lemma* justifying an implementation using Find with matching function Match:

```
Lemma Implement_Find :
  ∀ searchT Match f st R bag.
    R ≈ bag →
    P ⇜⇝ Match st →
      {l | l ≃ {x ∈ R | P x} } ⊇ Find bag st
```

The $\approx$ relation can simply be read as "contains the same elements as." The $\leftsquigarrow$ relation in the second hypothesis denotes that Match st is equivalent to P, in the sense that Match st returns true if and only if P holds:

$$P \leftsquigarrow f \equiv \forall x, P(x) \leftrightarrow f(x) = \mathit{true}$$

This lemma can be used to justify an implementation that exploits the underlying AVL tree's efficient lookup of tuples by their state attribute, as embodied in its Match function:

```
Method Enum (r : rep) (state : State) : rep × list ℕ:=
    ps ← Find r (Some state, λ_.true);
    ret (r, map (λp. p!pid) ps)
```

We can use the same lemma to derive an implementation of ByTime, although we wind up with rather inefficient code, since the Match method for our chosen data structure does not support efficient range queries:

```
Method ByTime (r : rep) (start end : ℕ) : rep × list ℕ:=
    ps ← Find r (None, λproc. start ≤? proc!date ≤? end));
    ret (r, map (λp. p!pid) ps)
```

An alternative selection of these parameters allows for set comprehensions defined by range predicates:

```
{ searchT := nat_opt × (nat_opt × nat_opt ) × (Tuple → bool);
  Match := λst, r. Match_EQState st_π1 r
                 ∧ Match_≤ time st_π2 r ∧ Match_⊥ st_π3 r }
```

Applying the refinement lemma with this Match function produces the desired implementation of ByTime:

```
Method ByTime (r : rep) (start end : ℕ) : rep × list ℕ:=
  ps ← Find r (None, (Some start, Some end), λ_. true);
  ret (r, map (λp. p!pid) ps)
```

In each of these three cases, the important insight needed is the mapping from the set-theory expression in the spec to the search-term parameter st used in the Find method. This relationship is captured in the P ⇜⇝ Match st hypothesis of Implement_Find; resolving the lemma's hypothesis without user interaction allows the method to be implemented automatically. Doing so in an extensible way allows new search

strategies to be added modularly. Conveniently enough, Ltac, Coq's tactic language, allows us to accomplish both tasks.

In actuality, using Implement_Find to implement ByTime produces an implementation with a "hole" (i.e. the unification variable ?) for the search term.

Method ByTime (r : **rep**) (start end : $\mathbb{N}$) : **rep** $\times$ list $\mathbb{N}$:= Find r ?

The missing search term also appears in a subgoal generated to satisfy the second assumption of Implement_Find, with appropriate instantiation of Match:

$\forall$ proc. start $\leq$ proc!time $\leq$ end $\leftrightsquigarrow$ Match ?

We resolve this goal by using Ltac to implement a backtracking search that determines ?. Conceptually, this search explores the space of possible search-term implementations by applying a series of lemmas showing how to match patterns in predicates with patterns of search terms. Figure 7 presents the rules for the $\mathsf{Match}_\perp$ function used by lists and the $\mathsf{Match}_{\mathsf{EQ}}$ function used by AVL trees. A crucial property of this AVL-tree encoding is that it is phrased as a *data-structure transformer*, extending any other data structure with an extra layer of efficient filtering by key equality, storing instances of the original data structure at the leaves of the AVL tree. We write $\mathsf{Match}_{\mathsf{EQ}_f,X}$ for the matching function of an AVL tree, keyed on field f, added on top of another data structure X.

The first three rules in Figure 7 provide some of the logic for splitting the work between the outer tree and the inner structures: MATCH$_\perp$, MATCH$_{EQ}$L, and MATCH$_{EQ}$R implement a predicate using the AVL tree, while the last rule, MATCH$_{EQ}$ALL, defers implementation to the nested structure. The rules in Figure 8, on the other hand, generically show how to implement conjunctions of predicates $P \wedge Q$ by decomposing the search into smaller subgoals. MATCH$_\wedge$ implements P and Q with the topmost and next-level Match functions; the assumption that v_2, the second component of the compound match term, is equivalent to $\lambda\_.\mathbb{T}$ ensures that it is safe to merge the search terms for P and Q by establishing that v_2 can never filter out a tuple. COMM$_\wedge$ and ASSOC$_\wedge$ reorder predicates to align better with the index structure. Figure 9 shows an example term derivation for our initial Enum implementation using these sets of rules.

Synthesizing new search terms is as simple as extending this backtracking search with appropriate rules. Figure 10 shows the collection of rules needed to support range trees. These rules closely mirror those in Figure 7: the first three rules show how to implement range predicates on an attribute f via the range tree's $\mathsf{Match}_{\leq f}$ function, while the last rule defers implementation to the nested data structure. Augmenting the search-term synthesis algorithm with these rules allows it to build the derivation for the implementation of ByTime using nested range trees illustrated in Figure 11.

Data-structure mutations can be implemented similarly by first applying a refinement rule implementing the mutation in terms of a $\mathsf{Bag}_{\mathsf{ADT}}$ method. As the semantics of each of these operations are also defined in terms of searchT and a Match function, each of these lemmas has a similar hypothesis, equating the set of mutated tuples to a search term, which can be dispatched automatically via the *same* set of rules used for queries. The task of implementing an ADT's methods is easily automated in Coq as a series of rewrites using these refinement rules, relying on an extensible set of proof rules

$$\frac{P \leftrightsquigarrow f}{P \leftrightsquigarrow \mathsf{Match}_\perp\ f}\ \textsc{Match}_\perp$$

$$\frac{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_X\ v_2}{\lambda x.\ x!f = v_1 \leftrightsquigarrow \mathsf{Match}_{\mathsf{EQ}_f,X}\ (\mathsf{Some}\ v_1, v_2)}\ \textsc{Match}_{EQ}\textsc{L}$$

$$\frac{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_X\ v_2}{\lambda x.\ v_1 = x!f \leftrightsquigarrow \mathsf{Match}_{\mathsf{EQ}_f,X}\ (\mathsf{Some}\ v_1, v_2)}\ \textsc{Match}_{EQ}\textsc{R}$$

$$\frac{P \leftrightsquigarrow \mathsf{Match}_X\ v}{P \leftrightsquigarrow \mathsf{Match}_{\mathsf{EQ}_f,X}\ (\mathsf{None}, v)}\ \textsc{Match}_{EQ}\textsc{All}$$

Figure 7. Constructing $\mathsf{Match}_\perp$ and $\mathsf{Match}_{\mathsf{EQ}}$ search terms

$$\frac{P \leftrightsquigarrow \mathsf{Match}_{X,Y}(v_1, v_2) \qquad \lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_Y\ v_2 \qquad Q \leftrightsquigarrow \mathsf{Match}_Y\ v_3}{P \wedge Q \leftrightsquigarrow \mathsf{Match}_{X,Y}(v_1, v_3)}\ \textsc{Match}_\wedge$$

$$\frac{Q \wedge P \leftrightsquigarrow f}{P \wedge Q \leftrightsquigarrow f}\ \textsc{Comm}_\wedge \qquad \frac{P \wedge (Q \wedge R) \leftrightsquigarrow f}{(P \wedge Q) \wedge R \leftrightsquigarrow f}\ \textsc{Assoc}_\wedge$$

Figure 8. Constructing search terms for compound predicates

$$\frac{\dfrac{\lambda\_.\mathbb{T} \leftrightsquigarrow \lambda\_.\mathsf{true}}{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_\perp\ \lambda\_.\mathsf{true}}\ \textsc{Match}_\perp}{\lambda \mathsf{proc.}\ \mathsf{proc!state} = \mathsf{state} \leftrightsquigarrow \mathsf{Match}_{\mathsf{EQ}_{\mathsf{state}},\perp}(\mathsf{Some\ state},\ \lambda\_.\ \mathsf{true})}\ \textsc{Match}_{\mathsf{EQ}}\textsc{L}$$

Figure 9. Deriving the search term for the Enum implementation using AVL trees.

$$\frac{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_X\ v_3}{\lambda x.\ v_1 \leq x!f \leq v_2 \leftrightsquigarrow \mathsf{Match}_{\leq f,X}\ ((\mathsf{Some}\ v_1, \mathsf{Some}\ v_2), v_3)}\ \textsc{Match}_\leq\textsc{B}$$

$$\frac{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_X\ v_2}{\lambda x.\ v_1 \leq x!f \leftrightsquigarrow \mathsf{Match}_{\leq f,X}\ ((\mathsf{Some}\ v_1, \mathsf{None}), v_2)}\ \textsc{Match}_\leq\textsc{R}$$

$$\frac{\lambda\_.\mathbb{T} \leftrightsquigarrow \mathsf{Match}_X\ v_2}{\lambda x.\ x!f \leq v_1 \leftrightsquigarrow \mathsf{Match}_{\leq f,X}\ ((\mathsf{None}, \mathsf{Some}\ v_1), v_2)}\ \textsc{Match}_\leq\textsc{L}$$

$$\frac{P \leftrightsquigarrow \mathsf{Match}_X\ v}{P \leftrightsquigarrow \mathsf{Match}_{\leq f,X}\ ((\mathsf{None}, \mathsf{None}), v)}\ \textsc{Match}_\leq\textsc{All}$$

Figure 10. Constructing $\mathsf{Match}_\leq$ search terms

$$\dfrac{\dfrac{\dfrac{\lambda\_.\mathbb{T} \rightsquigarrow \lambda\_.\mathsf{true}}{\lambda\_.\mathbb{T} \rightsquigarrow \mathsf{Match}_{\perp}\ \lambda\_.\mathsf{true}}\ \textsc{Match}_{\perp}}{\lambda \mathsf{proc.\ start} \leq \mathsf{proc!time} \leq \mathsf{end} \rightsquigarrow\ \mathsf{Match}_{\leq_{\mathsf{time}},\perp}(\mathsf{(Some\ start,\ Some\ end)},\ \lambda\_.\ \mathsf{true})}\ \textsc{Match}_{\leq}\mathrm{B}}{\lambda \mathsf{proc.\ start} \leq \mathsf{proc!time} \leq \mathsf{end} \rightsquigarrow\ \mathsf{Match}_{\mathsf{EQ}_{\mathsf{state}},\leq_{\mathsf{time}},\perp}(\mathsf{None},\ \mathsf{(Some\ start,\ Some\ end)},\ \lambda\_.\ \mathsf{true})}\ \textsc{Match}_{\mathrm{EQ}}\mathrm{ALL}$$

Figure 11. Deriving the search term for a ByTime implementation incorporating range trees

to automatically dispatch side conditions relating abstract operations to method parameters.

5. Data-Representation Implementation

We now turn to the final step of our synthesis process: the selection of concrete data structures, with rewriting of intermediate programs to use the operations of the chosen data structures. We decompose this task into two steps: first showing that the semantic interfaces selected in the previous step are *sufficient* to implement all abstract operations, before synthesizing an implementation of each interface. This decomposition is realized as a new honing tactic in the core Fiat library, finish sharpening with delegation $\overline{i}$ using $\approx_{\overline{i}}$, which uses a list of "delegate" ADT interfaces $\overline{i}$ and a parametric abstraction relation $\approx_{\overline{i}}$ to automate this natural decomposition of derivation steps. This tactic first shows that the current ADT implementation is appropriately parametric by iteratively replacing calls to methods of $\overline{i}$ with calls to corresponding implementations. As long as each $\mathsf{Bag}_{\mathsf{ADT}}$ method is called with the state returned by a previous method call, such rewrites are justified. This condition suffices to justify that the returned state of every method is a valid abstraction of the set returned by the original method call. The tactic then derives the final ADT implementation by simplifying the monadic structure until each method body is the ret of a value, standing for a singleton set. The tactic then generates a final proof obligation to give the set of concrete $\mathsf{Bag}_{\mathsf{ADT}}$ data structures $\overline{I}$ used in the final implementation of an ADT. Here is how we phrase such a deduction that formalizes a choice of concrete data structures:

$$\overline{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_i \gtrsim ?_i}$$

5.1 Extending the Data-Representation Toolbox

The previous version of the Query Structures library only considered the combinations of AVL trees and lists illustrated by Figure 12, which limited it to only supporting fast searches for sequences of specific attribute values. The AVL tree implements a map from a tuple attribute (e.g. state) value to another data structure (represented in Figure 12 by a dotted box) containing all the tuples with that key value. Adding new nested and leaf data structures to this toolbox allows us to tailor the available searches further: Figure 2, for example, presents the data structure built by nesting a range tree within an AVL tree. Keying the first layer of this data structure on state and the second layer on time lets us search efficiently for the precise set of tuples needed by both Enum and ByTime.

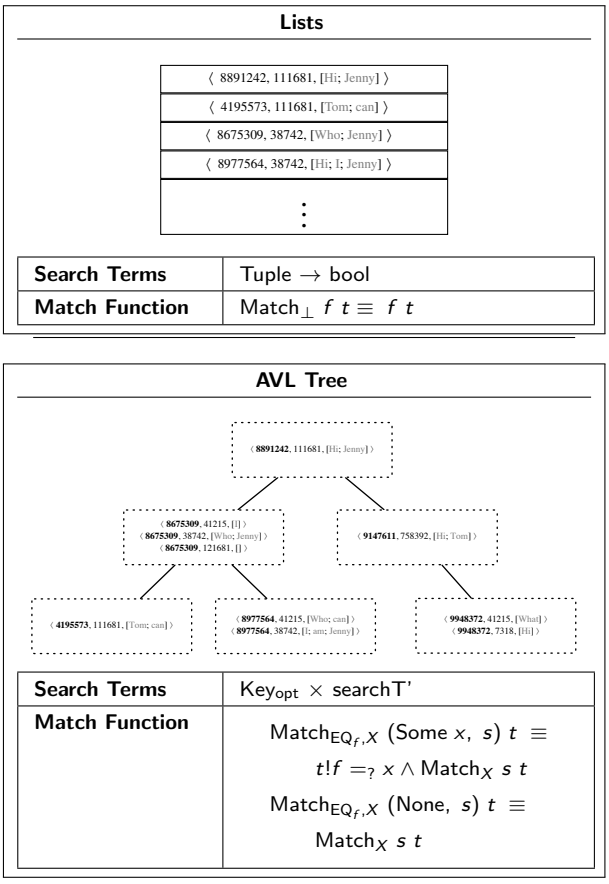


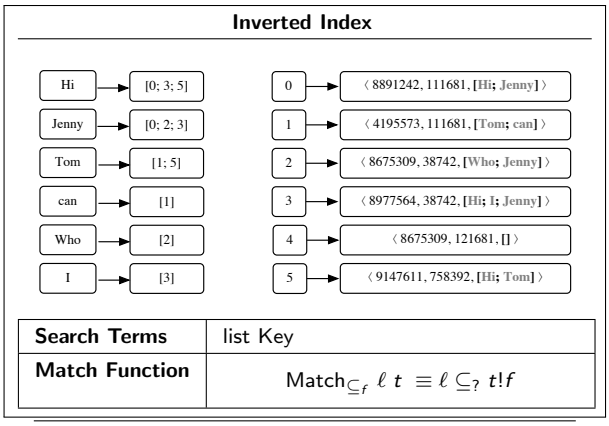
Figure 12. Building nested AVL trees

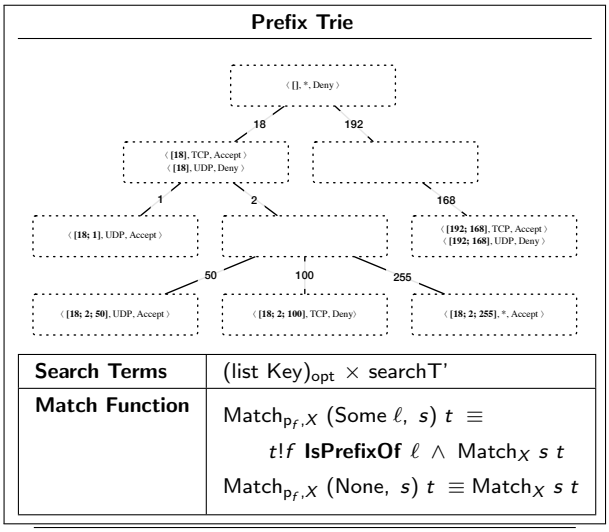
Figure 13 lists the three additional data structures we have incorporated into the framework, each of which supports a different kind of search: range trees can efficiently answer range queries, tries implement prefix matching, and inverted indexes quickly find keywords in a document.

Figure 12 and Figure 13 include the values of the searchT and Match parameters for each of the five data structures and data-structure transformers that we have built. Those parameters precisely characterize the efficient searches each supports. In order to incorporate a data structure X in our framework, we need to prove that it meets the $\mathsf{Bag}_{\mathsf{ADT}}$ interface for its Match_X function:

$$\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_X \gtrsim X$$

This proof can be developed interactively using Fiat's honing tactics, although these more specialized derivations resemble typical verification of functional data structures in a proof





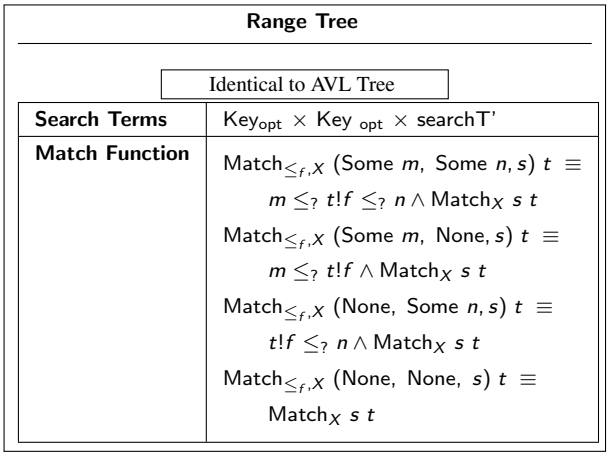


Figure 13. Data-structure extensions

assistant more than they do the Query Structures derivations we have presented so far.

Figure 14 gives the statements of correctness for each of our five data structures; as with the range-tree example, each nested data structure relies on a proof that the data structure used for its subnodes also implements the $\mathsf{Bag}_{\mathsf{ADT}}$

$$\frac{}{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_{\perp} \gtrsim \mathsf{List}_{\mathsf{I}}}\ \textsc{BagList}$$

$$\frac{}{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_{\subset} \gtrsim \mathsf{IIndex}_{\mathsf{I}}}\ \textsc{BagII}$$

$$\frac{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_X \gtrsim \mathsf{Impl}_X}{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_{\mathsf{EQ},X} \gtrsim \mathsf{AVL}_{\mathsf{I}}(\mathsf{Impl}_X)}\ \textsc{BagAVL}$$

$$\frac{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_X \gtrsim \mathsf{Impl}_X}{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_{\leq,X} \gtrsim \mathsf{Range}_{\mathsf{I}}(\mathsf{Impl}_X)}\ \textsc{BagRange}$$

$$\frac{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_X \gtrsim \mathsf{Impl}_X}{\mathsf{Bag}_{\mathsf{ADT}}\ \mathsf{Match}_{\mathsf{P},X} \gtrsim \mathsf{Trie}_{\mathsf{I}}(\mathsf{Impl}_X)}\ \textsc{BagTrie}$$

Figure 14. Implementation lemmas for available data structures

interface. As with our other synthesis tasks, using a collection of these lemmas to discharge the final obligation generated by the delegation honing tactic allows us to automatically and extensibly derive a correct implementation for each bag in a Query Structure. The following derivation using these rules implements the ProcTable table as a combination of AVL and range trees, for example:

$$\frac{\dfrac{\dfrac{}{\mathsf{Bag}_{\mathsf{ADT}}\ \ \mathsf{Match}_{\perp} \gtrsim \mathsf{IIndex}_{\mathsf{I}}}\ \textsc{BagList}}{\mathsf{Bag}_{\mathsf{ADT}}\ \ \mathsf{Match}_{\leq_{\mathsf{time}},\perp} \gtrsim \mathsf{Range}_{\mathsf{I}}(\mathsf{List}_{\mathsf{I}})}\ \textsc{BagRange}}{\mathsf{Bag}_{\mathsf{ADT}}\ \ \mathsf{Match}_{\mathsf{EQ}_{\mathsf{state}},\leq_{\mathsf{time}},\perp} \gtrsim \mathsf{AVL}_{\mathsf{I}}(\mathsf{Range}_{\mathsf{I}}(\mathsf{List}_{\mathsf{I}}))}\ \textsc{BagAVL}$$

This final step produces a complete ADT implementation of our original specification. Since each method is now a normal Gallina function, they can be extracted to OCaml using Coq's extraction mechanism.

5.2 Packaging Extensions

We pause here to review the extension points that allow new data structures to be incorporated to each phase of our approach to data representation synthesis.

1. The efficient searches supported by the new data structure must be characterized via a type of search terms and a Match function.
2. Hints associating search predicates with indexing strategies must be given in the form of $P \sqsupseteq \ell$ rules.
3. Implementation strategies for supported clause types must be provided in the form of $\mathsf{P} \rightsquigarrow \mathsf{Match}$ rules.
4. The new data structure must be proven to meet the $\mathsf{Bag}_{\mathsf{ADT}}$ interface for those values.

Figure 15 summarizes each of these ingredients for range trees, which have been threaded throughout the previous section, with a pointer in each case to the original figure.

We have realized the data-representation synthesis procedure outlined above as the new master_plan honing tactic, enabling an efficient implementation of one of our case stud-

Search Terms (Figure 13)	$\mathsf{Key_{opt}} \times \mathsf{Key_{opt}} \times \mathsf{searchT'}$
Match Function (Figure 13)	$\mathsf{Match}_{\mathsf{p}_f,X}\ (\mathsf{Some}\ n, \mathsf{None}, s)\ t \equiv$ $n \leq t!f \wedge \mathsf{Match}_X\ s\ t$
$\mathsf{Bag}_{\mathsf{ADT}}$ Implementation (Figure 14)	$\mathsf{Bag}_{\mathsf{ADT}}$ $(\mathsf{Key_{opt}} \times \mathsf{Key_{opt}} \times \mathsf{searchT}_X)$ $\mathsf{Match}_{\leq,X} \gtrsim \mathsf{Range}_{\mathsf{I}}(\mathsf{Impl}_X)$
Clause-Implementation Strategies (Figure 10)	$\lambda x.\ v_1 \leq x!f \leq v_2 \rightsquigarrow$ $\mathsf{Match}_{\leq_f,X}\ ((\mathsf{Some}\ v_1, \mathsf{Some}\ v_2), v_3)$
Data-Selection Hints (Figure 6b)	$(x!f \leq v) \sqsupseteq [\leq_f]$

Figure 15. Summary of extension ingredients (with range-tree examples)

ies that uses AVL trees, range trees, and inverted indexes to be derived by the following five-line proof script:

```
Definition SharpenedAlbum : SharpenedADT AlbumSpec.
  master_plan (InclusionIndexTactics ⊗ RangeIndexTactics ⊗
      EqIndexTactics).
Defined.
Definition AlbumImpl := Eval simpl in SharpenedAlbum_π1.
```

This tactic takes as input a plugin of implementation strategies including each of the four ingredients needed to incorporate a new concrete data structure into our approach. These plugins can be combined with the $\otimes$ operator. A small amount of implementation cleverness is needed to encode plugins properly in Coq's tactic language Ltac, where we must Church-encode records to get around Ltac's lack of native record support, but our extended Query Structure library also provides a generic packaging combinator.

6. Evaluation

Figure 16 summarizes the effort required to incorporate a new data structure for our framework, with each Ltac plugin taking about 15 minutes to implement and consisting of less than 250 lines of fairly boilerplate code. Building and verifying the three additional supporting functional data structures took roughly two person weeks total, but this is in line the effort required to verify a typical functional data structure. In exchange for that modest effort, we open up a new set of implementations for our automated master_plan tactic to consider during derivations.

Extension Type	Predicate	Data Structure	Ltac LoC
Range Queries	$m \leq ? \leq n$	Range Tree	251
Set Inclusion	$I \subseteq ?$	Inverted Index	168
Prefix Matching	p PrefixOf ?	Trie	215

Figure 16. Implemented extensions

To demonstrate the applicability of our new extensible query-planning tactic, we have applied the tactic to the six example programs listed in Figure 17. The first three examples of a bookstore, weather database, and a stock-market database are from the original Fiat paper and thus only contain that paper's equality clauses. The next example of a database of text messages and phone contacts uses a set-inclusion clause to search for messages containing a set of keywords. We also include a photo-album database with queries for photos containing a set of tagged people and for photos in a specified date range, which respectively use set-inclusion and range queries. The final example, packet classification, uses IP-address prefix matching to search a database of forwarding rules in order to identify the right policy for a packet.

Example	Equality	Range	Inclusion	Prefix
Bookstore	×			
Stocks	×			
Weather	×			
Messages	×		×	
PhotoAlbum	×	×	×	
Classifier	×			×

Figure 17. Examples and their clause types

We have extracted verified OCaml implementations of Messages and Classifier and benchmarked them using both the master_plan tactic with just AVL trees and lists and with a version extended with new data structures. The observed performance on an Intel Core i7 CPU @ 2.2 GHz is as expected, with a notable improvement for queries using the extra plugins. Figure 18 presents the runtimes of two such queries: the RelevantMessages query looks for any messages containing a set of keywords in the Messages Query Structure, while the Classify query uses prefix matching to find relevant forwarding rules. We evaluated the performance of 10000 random queries on databases populated by randomly generated tuples, varying the size of the database from 1000 to 20000 tuples. It is important to note that the baseline performance of Classify is poor because the table of rules does not have any indexes supporting this query, resulting in a method that is effectively implemented as a filter over a list.

Our case-study programs are in the directory

```
src/Examples/QueryStructure
```

in our supplement.

7. Related Work and Discussion

Data-representation selection has been considered in a number of different contexts, ranging from the initial investigations into the foundations of data-representation independence, to the automation of the choice of data structures and transformation. We combine work from all three areas to build the first automated and extensible approach for correct-by-construction data-representation selection that *invents new*

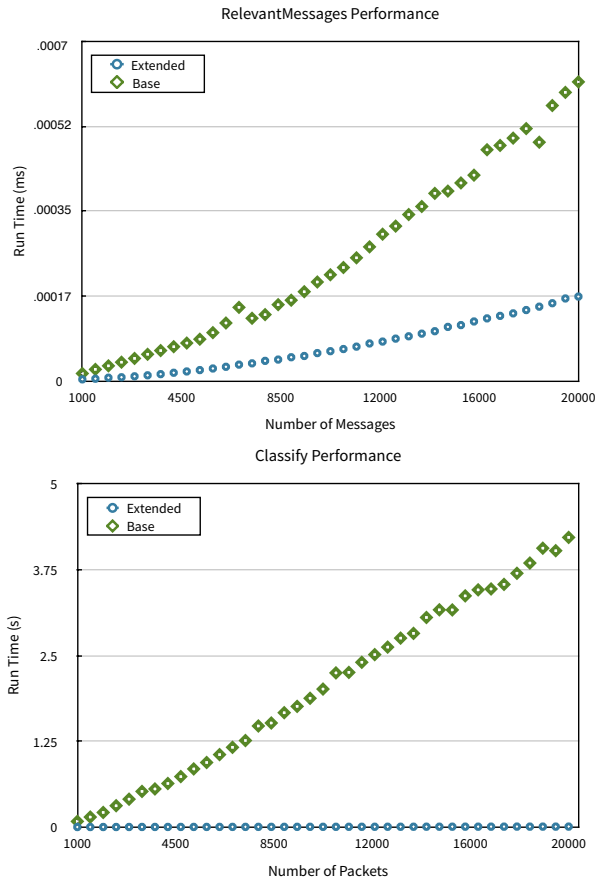


Figure 18. Total query time for vanilla and extended master_plan-derived Query Structures implementations

data structures on the fly and does *global analysis* of a spec to decide on the right data structure for its access patterns.

Automated Data-Representation Selection There were a number of early programming languages featuring sets as built-in language abstraction; the selection of efficient implementations for these abstractions was important for generating efficient code for these languages. The thesis of Low [14] presented an automated system for implementing programs defined using abstract sets by heuristically selecting a data structure drawn from a fixed library. Both the RAPTS [15] and SETL [16] languages specified programs at a high level using set-theoretic notation, using a compiler to derive efficient implementations. The final compilation step selected efficient implementations for sets of shared elements, but the engine was limited to a single data structure combining an array with a linked list. In both RAPTS and SETL, more complicated structures, e.g. trees, had to be programmed explicitly. Our new approach is not restricted to sets and, more importantly, does not try to select a general implementation of the set interface– we select an implementation of sets supporting the behaviors required by the operations of the ADT.

The more recent RELC compiler [5] for synthesizing implementations of abstract data types specified at a high level using abstract relational descriptions is perhaps philosophically closest to the approach presented here. From a high-level data-representation strategy and a set of functional dependencies, the RELC compiler synthesized an efficient C implementation supporting query and update operations. The compiler selects how to map high-level relational operations to a flexible key-value interface provided by the data-representation strategy. An optimal data representation is selected by iterating through the set of possible implementations strategies using an autotuner. Our semantic-interface inference is a more general solution to this problem, in that it is not constrained to relational specifications. Even within the Query Structures domain, we are not limited to equational where clauses.

The CHAMELEON tool [19] helps to select the best implementation of a fixed collection interface by instrumenting a JVM to profile the run-time behavior of a collection implementation, e.g. run time, memory usage, etc. An extensible set of rules is used to select a final implementation based on profiling information gathered from representative test runs. Where CHAMELEON provides profiling support, the COCO framework [25] dynamically identifies and replaces inefficient implementations of a fixed collection interface at runtime. We view both of these dynamic techniques are complementary to our approach, as run-time profiling could be used to enhance our operation selection heuristics and we can imagine synthesizing multiple data representations which could be adaptively selected at run-time.

Representation Transformation in a Proof Assistant Hoare [8] first introduced the notion of specifying and verifying algorithms at a high level using proof-oriented data abstractions and then transporting those algorithms and proofs to more efficient implementations via abstraction functions. He et. al [6] later extended this approach to use relations and nondeterminism, which has been realized in implementations in both Coq [3] and Isabelle [10, 11]. The latter tool, Autoref, automatically selects implementations of data abstractions in an extensible way, using transport rules showing how to implement operations on the abstract data for concrete data. Autoref automatically selects these concrete data implementations using a phase that explicitly annotates each term in an abstract expression with the relation that should be used to transport it. This annotation is carried out via a depth-first search that combines prioritized sets of abstraction relations and transfer rules to select the "best" implementation. This depth-first search effectively finds the best implementation for the *first* use of a data abstraction in a term, as opposed to our semantic interfaces, which allow for selection based on *every* occurence. Autoref also does not allow for the construction of ad-hoc data representations during this data-selection phase, disallowing our approach to on-demand construction of nested data structures.

In contrast to the above approaches which rely on user-supplied and verified transformation rules, the Jennisys [12] language attempts to automate the synthesis of implementations with minimal user interaction. As in Fiat, in Jennisys, classes are specified using an abstract model of internal state. Users then supply a concrete data implementation and a *coupling invariant* relating the abstract model and the concrete data representation. Given this invariant, Jennisys uses the Dafny [1] program verifier to explore the set of possible method implementations for the supplied data implementation; if an implementation is found, it is guaranteed to be correct. The high level of automation in Jennisys comes at a cost of limited applicability, and Jennisys does not currently support automatic selection of data implementations and coupling invariants.

Representation Transformations via Metaprogramming Both Racket's implementation of languages as libraries [23] and Scala's Lightweight Modular Staging (LMS) framework [18] rely on a combination of macros and syntax transformations to implement compilers that eliminate abstraction overhead, enabling programs to be written at a high level and then compiled to efficient implementations, or so-called "abstraction without regret." The key difference between these metaprogramming approaches and Fiat's is that each Fiat derivation produces a proof certifying that the derived code is correct. This frees users from trusting the metaprogramming framework, which in turns allows them to safely extend the framework with their own transformations without risking unsoundness.

The LegoBase query engine [9] utilizes Scala's LMS to compile query plans written in Scala into efficient C implementations, yielding queries that outperformed traditional systems in an experimental evaluation of queries drawn from the TPC-H benchmarks. LegoBase treats query planning (i.e. mapping semantic constraints to a data structure interface) and index (i.e. abstract data structure) selection as orthogonal problems; these are precisely the problems that our approach addresses in a more general setting. LMS has also been used to implement scoped data-representation transformations [24] in Scala using a "transformation object" encoding a map between high (abstract) and low (concrete) representations and a limited set of "pass-through" methods. Operations on low-level representations that are not in this "pass-through" set are implemented by transformations to and from the high-level representations. These transformation objects are user-supplied, whereas our concern is effectively the synthesis of the mapping, low representation, and complete set of pass-through methods.

Deductive Synthesis Kestrel Institute has used Specware [21], its deductive synthesis framework, to derive a number of complex correct-by-construction programs, including families of garbage collectors [17], SAT solvers [20], and network protocols. Specware derives programs through a series of user-guided steps, generating Isabelle/HOL obligations establishing the correctness of each step. The complex algorithms that Kestrel has focused on require highly manual derivations, which do not feature the same opportunities for extensible automation as the algorithmically "simple" domain of SQL operations presented here. The PECOS deductive synthesis system [2] relied on a database of pattern-matching rules to refine a node-based program representation. Users could extend the set of rules to include new implementation strategies, but as a result, data-structure selection was limited to those matching a fixed interface (with no first-principles proofs of correctness).

Database Engines Extensible indexing strategies have long been a design consideration for the Postgres database [22] and its successors. Early versions of the systems supported fast lookup of user-defined abstract datatypes implementing comparison operations supported by built-in index data structures. As a result, the vocabulary of search strategies was limited to equality and range queries via B+-Trees and equality, overlap, and containment queries via R-Trees. More recent versions support efficient queries with user-defined predicates that satisfy an abstract interface via generic implementations of indexed trees [7] and inverted indexes.

8. Conclusion

We have introduced the first proof-generating, automatic program derivation system that is **extensible with new recipes for customizing data structures on the fly and employing them to realize specifications**. Our system builds on the Fiat Coq framework and its core concepts of nondeterministic computations and abstract data types (ADTs). We formalize a general ADT of bags of tuples, which serves as the connection point between queries which assume the existence of appropriate bags, and data-structure implementations which implement the bag interface. The bag ADT is parameterized on a semantic characterization of a family of supported searches, and, given a library of such realizations, our tactics automatically crawl a specification to figure out which realizations will be useful in deriving it. Given the results of this analysis, we also automatically choose a nested composite data structure and rewrite the specification into an executable form that implements different parts of queries with the searching facilities of different constituent data structures. Extending the system with a new data structure is relatively easy, with a set of new heuristics being particularly simple to write, in a few hundred lines of stylized tactic definitions.

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(2):73–119, 1979.

[3] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In *Certified Programs and Proofs*. Springer International Publishing, 2013.

[4] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, 2015.

[5] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011.

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

[7] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Databases*, 1995.

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

[9] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, June 2014.

[10] P. Lammich. Automatic data refinement. In *Interactive Theorem Proving*. Springer Berlin Heidelberg, 2013.

[11] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer Berlin Heidelberg, 2012.

[12] K. R. M. Leino and A. Milicevic. Program extrapolation with jennisys. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, New York, NY, USA, 2012. ACM.

[13] B. Liskov and S. Zilles. Programming with abstract data types. In *Symposium on Very High Level Languages*, New York, NY, USA, 1974. ACM.

[14] J. Low. *Automatic Coding: Choice of Data Structures*. PhD thesis, Stanford University, August 1974.

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

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

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

[18] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. POPL '13, pages 497–510, New York, NY, USA, 2013. ACM.

[19] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, 44(6):408–418, June 2009.

[20] D. R. Smith and S. J. Westfold. Synthesis of propositional satisfiability solvers, 2008.

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

[22] M. Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the Second International Conference on Data Engineering*, pages 262–269, Washington, DC, USA, 1986. IEEE Computer Society.

[23] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. *SIGPLAN Not.*, 46(6): 132–141, June 2011.

[24] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating ad hoc data representation transformations. OOPSLA 2015, pages 801–820, New York, NY, USA, 2015. ACM.

[25] G. Xu. Coco: Sound and adaptive replacement of java collections. ECOOP'13, pages 1–26, Berlin, Heidelberg, 2013. Springer-Verlag.