The End of History? Using a Proof Assistant to Replace Language Design with Library Design*

Adam Chlipala[1], Benjamin Delaware[2], Samuel Duchovni[3], Jason Gross[4], Clément Pit-Claudel[5], Sorawit Suriyakarn[6], Peng Wang[7], and Katherine Ye[8]

1 **MIT CSAIL, Cambridge, MA, USA**
`adamc@mit.edu`
2 **Purdue University, West Lafayette, IN, USA**
`bendy@purdue.edu`
3 **MIT CSAIL, Cambridge, MA, USA**
`dukhovni@mit.edu`
4 **MIT CSAIL, Cambridge, MA, USA**
`jgross@mit.edu`
5 **MIT CSAIL, Cambridge, MA, USA**
`cpitcla@mit.edu`
6 **MIT CSAIL, Cambridge, MA, USA**
`sorawit@mit.edu`
7 **MIT CSAIL, Cambridge, MA, USA**
`wangp@mit.edu`
8 **Carnegie Mellon University, Pittsburgh, PA, USA**
`kqy@cs.cmu.edu`

Abstract

Functionality of software systems has exploded in part because of advances in programming-language support for packaging reusable functionality as libraries. Developers benefit from the uniformity that comes of exposing many interfaces in the same language, as opposed to stringing together hodgepodges of command-line tools. Domain-specific languages may be viewed as an evolution of the power of reusable interfaces, when those interfaces become so flexible as to deserve to be called programming languages. However, common approaches to domain-specific languages give up many of the hard-won advantages of library-building in a rich common language, and even the traditional approach poses significant challenges in learning new APIs. We suggest that instead of continuing to develop new domain-specific languages, our community should embrace library-based ecosystems within very expressive languages that mix programming and theorem proving. Our prototype framework Fiat, a library for the Coq proof assistant, turns languages into easily comprehensible libraries via the key idea of modularizing *functionality* and *performance* away from each other, the former via *macros that desugar into higher-order logic* and the latter via *optimization scripts* that derive efficient code from logical programs.

1998 ACM Subject Classification F.3.1 [Specifying and Verifying and Reasoning about Programs] Mechanical Verification, F.4.1 [Mathematical Logic] Mechanical Theorem Proving, Logic and Constraint Programming, I.2.2 [Automatic Programming] Program Synthesis

* This work has been supported in part by NSF grants CCF-1253229, CCF-1512611, and CCF-1521584; and by DARPA under agreement numbers FA8750-12-2-0293 and FA8750-16-C-0007. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

© Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye;
licensed under Creative Commons License CC-BY
2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Domain-specific languages, synthesis, verification, proof assistants, software development

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.3

1 The Case for Replacing Languages with Libraries in a Proof Assistant

As a programmer today, it is hard to imagine getting anything done without constant reuse of libraries with rather broad APIs. Complex production software systems weave together many different libraries hosted in a single language where integration is eased by a shared vocabulary of concepts like objects, functions, types, and modules. We can imagine piecing together similar end products from Frankenstein's monsters of distinct languages for different styles of programming, tied together with command-line tools and heroic build processes. Most of us are glad not to live in a world where that is the best option, however, considering several key advantages of the language-integrated approach.

- **Learnability.** To learn an arbitrary programming language, one might turn to its reference manual, which is prone to imprecision and likely to become out-of-date; or one might try to read the language's implementation, which unavoidably mixes in implementation details irrelevant to the user. In contrast, a library in a statically typed language inherits "for free" a straightforward characterization of the valid programs: precisely those that type check against the API, which is written out in a common formalism.
- **Interoperability.** With all libraries defining first-class ingredients that live in a common formalism, it becomes easier to code, for example, polymorphic operations that generalize over any such ingredients.
- **Correctness.** The developers of the chosen language implementation need to worry about getting it right, but then authors of individual libraries may rely on the language's encapsulation features to limit how much damage bugs in their libraries can inflict on other libraries and their private state.

All in all, we programmers can pat ourselves on the backs for collectively coming up with such a satisfyingly effective approach to building big things out of smaller, very general things. *However*, we claim there remain many opportunities to improve the story. *Domain-specific languages (DSLs)* are a paradigm growing in popularity, as programmers find that the API formalisms of general-purpose languages are not flexible enough to fit important packages of reusable functionality. Instead, a new notation is invented that allows much more concise and readable descriptions of desired functionality in a particular domain. DSLs have found widespread success in a range of domains, including HTML+CSS for layout, spreadsheet formulas for tabular data, and Puppet for system configurations. For programmer-facing tasks, DSLs such as SQL and BNF have been adopted widely, to the point that they are *the* standard solutions for interacting with databases and building parsers, respectively. Despite these isolated success stories, DSLs have not reached the ubiquity of library-based solutions in the average programmer's toolbox. Simply implementing a new DSL can require considerable effort, if one chooses to build a freestanding compiler or interpreter. Alternatively, we might have an *embedded* DSL that actually *is* a library within a general-purpose language, but which delineates its own new subset of that language with a rather distinctive look, perhaps calling library combinators to construct explicit abstract syntax trees. We claim both strategies introduce substantial friction to widespread adoption of DSLs.

- **Learnability.** A DSL with a freestanding implementation is just as unlearnable as a new general-purpose programming language, thrown at the programmer out of the blue. An embedded DSL may be easier to learn, by reading the static type signature that defines it, though we claim that often this type signature is complicated by limitations of the host language, and it almost never expresses semantics (in fact, in many cases, the host language is a dynamically typed Lisp-like language).
- **Interoperability.** Freestanding DSL implementations bring on all the pain that we congratulated ourselves on avoiding above. For instance, freestanding parser generators force the use of Makefiles or similar to coordinate with the main program, while the database query language SQL has notoriously bad coupling to general-purpose languages, for instance relying on APIs that present queries as uninterpreted strings and invite code-injection vulnerabilities. Embedded DSLs tend to appear to the programmer as their own fiefdoms with their own types of syntax trees, which rarely nest naturally within each other without giving up the performance from clever compilation schemes.
- **Correctness.** The kind of metaprogramming behind a language implementation is challenging even for the best programmers, and it is only marginally easier for compact DSLs than for sprawling marquee general-purpose languages. It may make sense to invest in producing a correct Java compiler with traditional methods, but the necessary debugging costs may prove so impractical as to discourage the creation of new DSLs with relatively limited scopes.

How could we reach back toward the advantages of libraries as natural APIs within a single host language? Our core suggestion in this paper is to modularize *functionality* away from *performance*. So much of programming's complexity comes from performance concerns. Let us adopt an extreme position on the meaning of the word, taking it even to encompass concerns of computability, where typically it is not sufficient to write an unambiguous description of desired program behavior; we must express everything algorithmically. Imagine, instead, that the programmer is liberated from concerns of performance, being able to write truly *declarative* programs that state *what* is desired without *how* to achieve it. Every part of a program is expressed with its most natural notation, drawing on the libraries that take over for DSLs, exporting notations, among other conveniences. However, each notation desugars to a common language expressive enough to cover any conceivable input-output behavior, even uncomputable ones.

At this point, the programmer has codified a precise specification and is ready to make it runnable. It is time to introduce *performance* in a modular way. We follow the tradition of program derivation by stepwise refinement [8], but in a style where we expect the derivation process to be automatic. Every program mixes together derivation procedures drawn from different libraries, with each procedure custom-designed to handle the notations of that library. The end result is a proof-producing *optimization script* that strings together legal moves to transform specifications into efficient executable code. By construction, no optimization script can lead to an incorrect realization of an original program/specification.

This style promotes **learnability** with self-documenting macro definitions that are readable by the programmers who apply the macros, with each definition desugaring a parsing rule into a common higher-order logic, written for clarity and without any concern for performance; **interoperability** by the very use of that logic as the common desugaring target[1]; and **correctness** by codifying legal moves to transform specifications toward efficient code in optimization scripts.

[1] A caveat on the interoperability front, considering a popular alternative reading of that word, is that we are not primarily concerned with integrating with legacy systems. We are happy to design a clean-slate platform where libraries interface nicely, assuming that they commit to a new style of design.

Assuming one buys into this pitch, what would be the ideal platform for unifying all the ingredients? One proposal would be to leverage an existing language with metaprogramming features. Indeed, Racket's implementation of languages as libraries [38] and Scala's Lightweight Modular Staging (LMS) framework [32] enable programs to be written at a high level and then compiled to efficient implementations in order to achieve "abstraction without regret" through a combination of macros and syntax transformations. While both these approaches come close to achieving our goal, they both require that initial programs be executable, thereby imposing some algorithmic requirements on them, and they require the user to trust that the metaprograms implementing transformations are semantics-preserving.

In order to enable programmers to focus on *how* and not *what*, we propose using a very expressive logic and a macro system that desugars into it as the host language. We also need a way to code heuristics for transforming programs in that logic. In other words, we have a two-level language, with an object language of specifications and a metalanguage for manipulating specifications. The metalanguage should work in a correct-by-construction way, where we can be sure that no transformation breaks program semantics. The combination of these features enables a reimagination of embedded DSLs to have clean, declarative semantics unpolluted by concerns of executability, but which still result in efficient implementations.

Many readers will not be surprised at this point that we have described the core features of modern proof assistants! Such popular ones as Coq and Isabelle/HOL fit the bill; we chose Coq. All of the widely used proof assistants have their rough edges today, but we do imagine a future where more polished proof assistants serve as the IDEs of everyday programming in this style (and we hope that our experiments can help identify the best ways to go about that polishing). Our prototype framework **Fiat** [7] already makes it possible to code interesting programs inside of Coq, with extensibility plus strong separation of functionality from performance, generating assembly code with a proof of conformance to original functionality specifications. In the rest of this paper we review the core of Fiat, give some old and new examples of notation domains therein, and indulge in more philosophizing and speculation than we usually would in a conference paper.

2 A Flexible Core Language for Declarative Programs

The heart of the Fiat concept is a unified, flexible language for writing out the functionality of programs. We write these declarative programs using macros defined by domain libraries, but each macro has a simple syntax-mapping rule, so macros cannot be used directly to encode complex logic. Instead, we need a core language under the hood that we believe can be used to encode any reasonable program specification. Rather than reinventing the wheel, we start from Gallina, the logic of our favorite proof assistant Coq, which is already well-known from successful mechanized proofs from algebra [10] to compiler correctness [26].

However, the original program is not the end of the story, and there are reasons to add more superstructure on top of Gallina. We transform initial declarative programs gradually into efficient executable programs, and it is helpful to maintain the same core language for original programs, intermediate programs, and final executable programs. For that purpose, we chose the *nondeterminism monad*, also used in concurrent work by Lammich [21] on stepwise refinement in a different proof assistant.

We define our type family $\mathcal{P}$ of *computations* using the pun that the familiar notation for the powerset operator may also be thought of as standing for "program." The standard monad operators are defined as follows and can be proved to obey the monad laws, in terms

of the standard semantics of the set-theory operators we employ.

$$\begin{aligned} \texttt{return} &: \forall \alpha.\ \alpha \to \mathcal{P}(\alpha) \\ \texttt{return} &= \lambda x.\ \{x\} \\ \texttt{bind} &: \forall \alpha, \beta.\ \mathcal{P}(\alpha) \to (\alpha \to \mathcal{P}(\beta)) \to \mathcal{P}(\beta) \\ \texttt{bind} &= \lambda c_1, c_2.\ \bigcup_{x \in c_1} c_2(x) \end{aligned}$$

We introduce the usual shorthand $x \leftarrow c_1; c_2$ for `bind` c_1 $(\lambda x.\ c_2)$. Now we can write a variety of useful (though potentially non-executable) programs that periodically pick elements from mathematical sets. For instance, here is a roundabout and redundant way to express the computation of any odd natural number.

```
a ← {n ∈ ℕ | ∃ k ∈ ℕ. n = 2 × k};
b ← {m ∈ ℕ | ∃ k ∈ ℕ. m = 1 + 2 × k};
return (a + b)
```

Similarly, here is how one might compute the sum of the integer zeroes of a polynomial:

```
zs ← {xs ∈ list ℕ | NoDuplicates xs ∧ ∀ x, P(x) = 0 ⇔ x ∈ xs};
return (foldl (+) 0 zs)
```

More ominously, here is a program (referring to the set $\mathbb{B}$ of Booleans) that we should probably not try too hard to refine into an executable version.

```
b ← {b ∈ 𝔹 | b = true ⇔ P = NP};
if b then return 42
else return 23
```

This example also illustrates the *relational* character of the nondeterminism monad: defining computations using logical predicates makes it trivial to integrate potentially uncomputable logical functionality with standard functional programming. For instance, we use the normal `if` construct of Gallina, rather than defining our own. Such natural integration may even lull the programmer into a false sense of security, as in the above example, where choosing a branch of a conditional requires resolving a major open question in theoretical computer science! (It is at least fairly straightforward to formalize the proposition "P = NP" in a general-purpose proof assistant like Coq.)

We choose the superset relation $\supseteq$ as our notion of refinement between computations. We also sometimes read $c_1 \supseteq c_2$ as "c_2 implements c_1." In general, c_2 should be more algorithmic or more performant than c_1, and we chain together many such steps on the path to a final efficient program. For instance, by this definition, our example with odd numbers refines into `return` 7, because $\{n \mid n \text{ is odd}\} \supseteq \{7\}$.

It is also crucial that we have effective tools for rewriting in computations, since rewriting is a convenient way to structure refinement steps. Theorems like this one justify the use of standard rewriting rules:

$$\forall \alpha, \beta, c_1 : \mathcal{P}(\alpha), c_1' : \mathcal{P}(\alpha), c_2 : (\alpha \to \mathcal{P}(\beta)).\ c_1 \supseteq c_1' \Rightarrow \texttt{bind}\ c_1\ c_2 \supseteq \texttt{bind}\ c_1'\ c_2\,.$$

Applying this theorem with the fact $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N}.\ n = 2 \times k\} \supseteq \{4\}$ lets us refine the odd-number example above into this form:

```
a ← return 4;
b ← {m ∈ ℕ | ∃ k ∈ ℕ. m = 1 + 2 × k};
return (a + b)
```

From here, the monad laws allow us to remove the `bind` for a, substituting the value 4 for bound occurrences of a. Instead of simplifying, we could also apply an analogous theorem that justifies rewriting under binders, in the second term argument of `bind`.

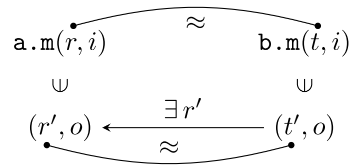

Figure 1 Refinement preserves similarity of internal values.

The nondeterminism monad provides a concise language for capturing a program's algorithmic content. To provide a complete declarative programming language, however, we need to add in the second element of the classic equation "programs = algorithms + data structures." To finish the story, Fiat also supports *data refinement* [14], allowing programs to operate over an abstract data model, introducing efficient data structures as part of the refinement process via an abstraction relation [15]. Consider the following declarative program, which filters out any number that is at least 100 in a set `s`:

```
return s ∩ {n | n < 100}
```

One reasonable implementation of this program replaces sets with splay trees and uses their `split` operation to perform the filter:

```
return fst(split(s, 100))
```

These two programs clearly return similar results, in the sense that the splay tree produced by the latter has the same elements as the set produced by the former, assuming the initial data had this relationship. We can get more formal with the following abstraction relation: $s \approx t \triangleq \forall n.\ n \in s \leftrightarrow n \in \texttt{elements}(t)$. Parameterizing the refinement relation over such relations captures this notion of similarity between a declarative program, succinctly stated with datatypes that are more abstract, and an implementation, which uses optimized data structures.

Under this approach, the behavior of an implementation depends on the abstraction relation used during refinement. As an example, every data type in the specification could be related to the unit type, to produce a valid but uninteresting refinement. In order to understand a refined program, a programmer cannot simply examine its initial specification – it is also necessary to consider the specific abstraction relation that produced that implementation. This requirement contradicts our proposed *learnability* criterion. It is also inherently antimodular, as any client of a refined program needs to be refined via the same abstraction relation. In order to enable modular reasoning while still permitting data-structure optimizations, Fiat exploits the familiar notion of data encapsulation provided by abstract data types (ADTs).

An ADT packages an internal *representation type* and a set of operations that build and manipulate values of that type. Fiat restricts data refinements to the representation types of ADTs. Clients are unable to observe the choice of data structure for an ADT's representation type thanks to *representation independence*, freeing an implementation to optimize its representation as it sees fit. In Fiat, an ADT specification uses an abstract model for its representation type and has operations that live in the nondeterminism monad, while an implementation is a valid refinement under an abstraction relation that is limited to optimizing the representation type. Intuitively, every implementation of an operation takes similar internal values to similar internal values, and its observable outputs are elements of the original specification, as illustrated by Figure 1. Thus, in contrast to other refinement

frameworks that allow arbitrary data refinements [6, 22], in Fiat a client can understand the behavior of a refined ADT just by looking at its specification.

3 Integrating DSLs

To demonstrate the flexibility of this approach in action, we present the development of a simple packet filter in Fiat. At a high level, such a filter has two components: decoding the "on-the-wire" binary packet into an in-memory representation and then consulting a set of rules to decide whether to drop or forward the packet. Each of these two algorithmic tasks can be expressed using a domain-specific language implemented as a Fiat library; we begin with a brief overview of the two appropriate libraries.

The domain of the first library is the decoding of bitstrings into high-level in-memory datatypes, compatibly with some specified binary format. For simplicity, we consider deterministic binary formats, where every datatype has a single valid encoded representation. In this setting, a format can be captured as a function from the original datatype to its encoding, as in the following encoder for a record with two fields:

```
T ≜ {A : string, B : list int}
encode (t : T) ≜ encodeInt(len(t!B)) ++ encodeString(t!A) ++ encodeList(t!B)
```

This format combines together existing encoders for strings, lists, and integers, using the last to encode the length of the list in `t.B`, which the decoder will need to decode this field correctly. Given such a format, the specification of a decoder is straightforward:

```
decode (s : BitString) ≜ {t | encode(t) = s}
```

Here we have used the nondeterminism monad to capture the fundamental correctness condition succinctly for a binary decoder. The library also supports derivation of such an implementation via conditional refinement rules, examples of which are given in Figure 2. Each rule is a theorem in higher-order logic, and, to derive a particular decoder automatically, the optimization script chains together rule applications in rewriting style (with the crucial consequence that applying optimization scripts cannot lead to incorrect programs). The first two rules decode the head of the bitstring before decoding the rest under the assumption that some projection f of the encoded datatype is equal to the decoded value. Subsequent derivation steps can make use of this information, e.g. the correctness of DecodeList depends on a previously decoded length value. The final rule, FinishDecoding, is used to finish a derivation when enough information has been decoded to determine the original datatype uniquely. An implementation of a decoder can be derived automatically using these (generic) rules, plus a rule for decoding integers:

```
  {t | encodeInt(len(t.B)) ++ encodeString(t.A) ++ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in                                      (DecInt)
  {t | len(t.B) = n ∧ encodeString(t.A) ++ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in      (DecString)
  {t | len(t.B) = n ∧ t.A = a ∧ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in      (DecList)
  let (l, s) = decodeList(s, n) in {t | len(t.B) = n ∧ t.A = a ∧ t.B = l ∧ [] = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in      (FinishDec)
  let (l, s) = decodeList(s, n) in if s = [] then {A ≜ a; B ≜ l} else fail
```

The key takeaways here are: given a binary format, writing an initial, declarative decoder is immediate and obvious, and while its implementation is more complicated, the correctness of

$$\frac{}{\{\texttt{t} \mid \texttt{P}(\texttt{t}) \land \texttt{encodeString}(\texttt{f}(\texttt{t})) +\!\!+ \texttt{s}' = \texttt{s}\} \supseteq \begin{array}{l}\texttt{let}\ (\texttt{v},\texttt{s}) = \texttt{decodeString}(\texttt{s})\ \texttt{in} \\ \{\texttt{t} \mid \texttt{P}(\texttt{t}) \land \texttt{f}(\texttt{t}) = \texttt{v} \land \texttt{s}' = \texttt{s}\}\end{array}}\ (\textsc{DecodeString})$$

$$\frac{\forall \texttt{x}.\ \texttt{P}(\texttt{x}) \to \texttt{len}(\texttt{f}(\texttt{x})) = \texttt{n}}{\{\texttt{t} \mid \texttt{P}(\texttt{t}) \land \texttt{encodeList}(\texttt{f}(\texttt{t})) +\!\!+ \texttt{s}' = \texttt{s}\} \supseteq \begin{array}{l}\texttt{let}\ (\texttt{v},\texttt{s}) = \texttt{decodeList}(\texttt{s},\texttt{n})\ \texttt{in} \\ \{\texttt{t} \mid \texttt{P}(\texttt{t}) \land \texttt{f}(\texttt{t}) = \texttt{v} \land \texttt{s}' = \texttt{s}\}\end{array}}\ (\textsc{DecodeList})$$

$$\frac{\forall \texttt{x}.\ \texttt{P}(\texttt{x}) \leftrightarrow \texttt{x} = \texttt{v}}{\{\texttt{t} \mid \texttt{P}(\texttt{t}) \land [] = \texttt{s}\} \supseteq \texttt{if}\ \texttt{s} = []\ \texttt{then}\ \texttt{v}\ \texttt{else}\ \texttt{fail}}\ (\textsc{FinishDecoding})$$

Figure 2 Refinement rules for deriving binary decoders.

```
empty        ≜ ∅
For x in i b ≜ table ← {l | i ∼ l};
                 fold_R (λ a b ⇒ l ← a; l' ← b; return (l ++ l'))
                   (return []) (map (λ x ⇒ b) table)
Where P b    ≜ {l | P → l ∈ b ∧ ¬ P → l = []}
Return a     ≜ return [a]
Count b      ≜ results ← b; return length(results)
```

Figure 3 Notations for querying sets (relation ∼ constrains a list to contain some permutation of the elements of a set).

one built by an optimization script is guaranteed by (proof-producing) refinement. Note also that the process is extensible without expanding the trusted code base, in that incorporating a decoder for a new type is as simple as writing a new decoder and proving the corresponding refinement rule.

The next library used in our packet filter is a DSL for writing SQL-like programs. The notations provided by this library, examples of which are shown in Figure 3, desugar into basic set- and list-comprehension operations. The `Where` notation showcases the extensibility provided by our core framework, as a clause uses an arbitrary predicate to filter the set in contrast to, say, SQL. Consider a declarative function that finds the size of an island in a set:

```
island ≜ {name : string, size : int, temp : int}
islands : set of island
sizeOf (name) ≜ For i in islands Where i!name = name Return i!size
```

Just as in SQL, the notation provides for a concise description of both the program and its functionality, as it desugars into an expression using familiar set operations. Also as with SQL, the key challenge in executing this program is selecting data structures supporting the needed searches. A user of this library can write out only the abstract model of the representation type of an ADT and then rely on the optimization script to solve this implementation challenge via data refinement. A pleasant consequence of encapsulating the sets inside the ADT's representation type is that "whole-program analysis" becomes possible: we can write optimization scripts that examine exactly the queries (and updates) that we have exposed, automatically tailoring a data representation to efficient execution of those operations. Our relational-data library does just that, using plugins to incorporate user-provided (and user-proved) data-structure strategies, relying on the correctness guarantees provided by the core of the framework to ensure that they preserve the functionality of the original programs.

These two libraries demonstrate how our approach promotes *learnability* by enabling concise, declarative specifications of functionality, while also maintaining *correctness* in the

face of extensibility via a machine-checked refinement trail. To round out our wish list with *interoperability*, we can see that they also play nicely with each other by combining them together to build a packet filter:

```
packet ≜ {src : word, name : list string, qtype : int}
rule   ≜ {name : list string, qtype : int, approve : boolean}
rules  ≜ set of rule
decide (s : BitString) ≜ p ← {p : packet | encodePacket(p) = s};
                         ans ← For r in rules
                                 Where r!name isPrefixOf p!name
                                 Where r!qtype = p!qtype
                                 Return r!approve;
                         return (head ans)
```

This example mixes the notations of the two libraries with normal functions (e.g. `head`), and it uses a custom `isPrefixOf` predicate in the `Where` clause of the query. More importantly, the optimization script that produces an implementation is also able to mix the implementation strategies provided by the libraries to handle the implementation tasks in both domains, automatically synthesizing the decoder for packets and selecting a data structure that supports prefix queries (tries, in this case).

4 Related work

There is a long history [8, 20, 29, 1] of using program transformations and stepwise refinement to obtain correct-by-construction, efficient implementations from specifications (albeit not necessarily in an automated fashion). Recent developments differ in guarantees obtained about the refined programs, intended application domains, degrees and styles of automation, and extensibility. Similarly, there is a rich line of academic work [12, 16, 2, 40] on the design and applicability of domain-specific languages: in fact, most early programming languages had domain-specific roots before they grew into general-purpose languages (LISP, the *list processor* for symbolic manipulations and AI; COBOL, the *common business-oriented language*; and FORTRAN, the *formula translator* for numerical computations). The following is a limited sampling of tools closely related to Fiat.

Stepwise Refinement Frameworks

The family of tools encompassing KIDS, DTRE, and Specware [34, 3, 37] allows users to decompose high-level specifications progressively into more and more concrete subproblems, until a concrete implementation can be supplied for each subproblem. The refinement style is similar to the one used by Fiat, with the main differences in how refinement steps are justified (Fiat is embedded in Coq and transparently exports a Coq proof obligation, while Specware relies on trusted proof-obligation generators to produce Isabelle/HOL goals justifying each transformation), target languages (Specware uses unverified transformations to extract C code, while the original Fiat system produces executable Gallina programs), composability (Fiat programs can be integrated into larger software developments verified in Coq), sound extensibility (Fiat tactics are proof-producing programs that run no risk of introducing unsoundness), and application domains (Fiat is mostly used for “simple” domains that lend themselves well to DSL development and admit clear specifications, allowing for a single refinement script to cover a large fraction of all programs expressible in the corresponding

DSL; Specware, on the other hand, has been used to synthesize correct-by-construction collections of complex algorithms, such as garbage collectors [30] or SAT solvers [35]).

Leon [19] is a deductive synthesis framework for deriving verified recursive functions on unbounded data types. Leon combines built-in recursion schemas, exhaustive enumeration, and counterexample-guided synthesis to generate the bodies of functional programs according to formally expressed postconditions. When the implementation chosen by the system is correct but not satisfactory, Leon users have the option to step in and perform refinement steps (verified refactorings) manually. Fiat has also been used to synthesize recursive programs [11] and uses less general automation: instead of a single synthesizer intended to cover all possible programs, Fiat specifications are refined using domain-specific optimization scripts that usually employ mostly deterministic strategies without backtracking. Users are free to introduce new rewriting steps and refinement strategies to achieve the desired performance characteristics.

Cohen et al. [6] used a notion of data refinement close to that of Fiat to develop and verify a rich algebra library in Coq: starting with high-level definitions written using "proof-oriented" data structures amenable to simple verification, the authors use data refinement to obtain an implementation with more efficient data structures satisfying the same guarantees. Our approach is different, in that we start from a potentially noncomputational, nondeterministic specification, which we refine to an implementation. We furthermore restrict data refinements to the representation types of ADTs, obviating the need for transporting proofs across an entire program.

Data-Structure Synthesis and Selection

Automatic data-structure selection was pioneered in SETL [33], a high-level language where programmers manipulate sets and associative maps through high-level primitives such as comprehensions and quantifiers, without committing to specific implementations of the underlying data structures. Instead, the SETL compiler employs a sophisticated static analysis to make concrete data-structure choices. Fiat's decoupling of specifications and performance yields a similar process. Unlike SETL, Fiat imposes no restrictions on the kind of data structures that can be used, the ways they can be combined, and the type of hints that programmers can give to the compiler to nudge it towards specific implementations. Fiat's sound extensibility makes it possible to substitute newly verified data structures at any step in the refinement.

More recently, Loncaric et al. [28] have built Cozy, a system for efficiently synthesizing a broad range of data structures, starting from a restricted DSL of data-retrieval operations and generating efficient object-oriented code using counterexample-guided inductive synthesis. Cozy synthesizes a high-level functional implementation of each operation using exhaustive enumeration augmented with a cost model to prune the search space, and from there deduces a good data representation, optionally using real-world benchmarks to autotune the selection. Though there are close similarities between the input language of Cozy and Fiat's SQL-style application domain, Fiat uses a mostly deterministic domain-specific compiler and hand-verified refinements instead of exhaustive enumeration and a general-purpose verifier. Fiat's SQL-style domain can in a sense be seen as an extensible proof-producing query planner, with strong extensibility granted by integration in a proof assistant. This vision provides an alternative answer to one of Cozy's original motivations, replacing unpredictable and hard-to-extend SQL engines.

Closely related to Cozy is Hawkins et al.'s RelC synthesizer [13], which decouples the relational view of the data from its in-memory representation (specified as a combination of

basic data structures such as hash tables, vectors, or linked lists) by automatically deriving low-level implementations of user-specified relational accessors and mutators compatible with the chosen representation. Fiat has a similar input language but provides stronger correctness guarantees and allows for proof-producing extensions to the existing compilation logic (Fiat optimization scripts cannot perform unsound transformations). Fiat is additionally an open-ended system, allowing users to combine multiple DSLs and use their respective compilers to synthesize parts of a larger verified program covered by end-to-end guarantees.

Leino and Milicevic [25] proposed dividing the effort of programming a verified component into three parts: a public interface providing a mathematical model of the object; a data-structure specification describing the layout and invariants of the underlying implementation; and an executable implementation of the data structure. Jennisys, a prototype implementation of this idea, allows users to synthesize the code part of a component automatically by extrapolating from pre- and postcondition-conforming input and output examples generated using the Dafny [24] program verifier. Fiat shares some of Jennisys' synthesis objectives but applies to different domains, does not commit to specific data layouts and implementation details, and rejects the traditional regime of dividing a program into data structures and algorithms (phrasing problems in terms of functionality and performance).

Domain-Specific Synthesis

The binary encoders and decoders presented in Section 3 are similar in spirit to programs written using bidirectional lens combinators in the Boomerang [4] programming language. A single Boomerang program represents a pair of transformation functions between source and target domains. Compiling a Boomerang program produces both a map from source to target and an inverse function guaranteed to propagate changes from a target back to the generating source object.

Many domains beyond the ones that we have focused on are amenable to our approach. SPIRAL [9] is a framework for automatically deriving high-performance digital signal-processing code. Bellmania [17] is a recent framework for deriving cache-efficient implementations of divide-and-conquer algorithms. Bellmania uses a unified formalism to encompass both relatively high-level specifications of dynamic programs and their low-level implementations, allowing programmers to derive cache-efficient code through expert application of trusted *solver-aided tactics*, a carefully crafted set of built-in program transformations. Bellmania uses an SMT solver to ensure that each tactic is used soundly and to assist users by synthesizing code fragments from concrete inputs and traces.

Domain-Specific Languages

More broadly, there is a large body of work on DSL design and implementation. Leijen and Meijer [23] introduce and highlight the advantages of embedding DSLs in higher-order typed languages. Kats and Visser have developed the Spoofax [18] language workbench, a metaprogramming framework encompassing DSL parsers, compilers, and IDE support. Tobin-Hochstadt et al. [38] used Racket to implement the high-performance Typed Racket language. Van der Storm et al. [39] use *object grammars* to define compositional DSLs.

5 Discussion and Future Directions

Many past systems have done principled generation of code from specifications, using either combinatorial search (e.g., with a SAT solver in Sketch [36]) or deductive derivation (e.g.,

with Specware [37]). What is the secret sauce that distinguishes Fiat from these past systems? We claim it is the careful combination of *correct-by-construction automation* with *manual design of abstractions and decomposition of programs into modules.* Fundamentally, Fiat is a refinement of today's standard wisdom in software development: there is no silver bullet for solving all design and implementation problems. Instead, developers need to work hard to design proper abstractions (e.g., classes, libraries). In the best case, many abstractions are highly reusable. However, when working in an unfamiliar programming domain, we expect to develop a few new abstractions, probably in concert with reusing many familiar ones. Fiat is not a program-synthesis system that generates code automatically from specifications in a fixed domain. Such systems have inherent limitations and are unlikely to scale in isolation to the full software-development problem. At the same time, Fiat is not a manual-derivation system in the style of Specware. Instead, Fiat embodies a new style of modular program decomposition, where some modules are similar to traditional programs, though they support higher-order logic in place of algorithmic constructs; while other modules are more unusual, implementing automated strategies for deriving good code from the other modules. The programmer still faces a difficult and manual task in decomposing a program in this way, but the principled use of formal logic and correct-by-construction rewriting dramatically dampens the traditional pain points that we have emphasized throughout this paper.

We hope that the Fiat approach or one like it can earn a place on the standard list of abstraction and modularity techniques for practical programming. The central idea is to allow separate coding of the *functionality* and *performance* parts of a program, which we see as a natural evolution of the implementation/interface distinction of data abstraction [27]: the interface becomes the declarative program itself, one that is specific enough that we are happy with any compatible implementation, which we then derive automatically with a short optimization script that soundly combines nontrivial procedures from libraries. Of course, remembering all of the folk stories of genies run amok when their users wish incautiously, it is a tall order to design a specification discipline that minimizes unintended consequences. At a minimum, the technique needs to be extended with performance requirements as part of functionality, and no doubt some aspects of security should be added explicitly, too, though many of them are implied by functional correctness.

Our ongoing work gives library authors broad discretion in crafting high-performance optimization strategies by connecting to a proof-carrying-code system [5], admitting optimization rules that refine functional programs into assembly code, in concert with requirements to link against handwritten, low-level, verified implementations of imperative data structures [31]. We are also thinking about and prototyping a number of other domains with simple declarative starting points and effective correct-by-construction optimization strategies: textual formats specified by context-free grammars, SMT-style solvers specified by logical theories, and optimized big-integer cryptographic primitives specified by whiteboard-level math. There also seems to be no shortage of more far-out ideas that fit into the framework. We would like to, for instance, replace `make` and other build systems with use of a Fiat-style framework. Instead of writing "compile `a.c` into `a.o` using `gcc`," the build configuration would read "choose an element of the set of object files meeting a fixed semantic contract with the following C AST." That is, a (verified) compiler is just a relatively predictable kind of optimization script. Combinators could be used to mix together all such directives into build specifications for whole projects, oriented toward proving top-level project theorems, protecting against bugs in a variety of internal development tools.

Acknowledgments. The first author was inspired to start a project in this research area by conversations with Daniel S. Wilkerson and Simon Goldsmith about their unpublished work. For their helpful feedback, we thank the anonymous SNAPL reviewers and our shepherd Jean-Baptiste Tristan.

References

1. David R. Barstow. Domain-specific automatic programming. *IEEE Softw.*, 11(11):1321–1336, November 1985. `doi:10.1109/TSE.1985.231881`.
2. Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, August 1986. `doi:10.1145/6424.315691`.
3. Lee Blaine and Allen Goldberg. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, pages 165–204. Elsevier, 1991.
4. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proc. POPL*, pages 407–419, 2008. `doi:10.1145/1328438.1328487`.
5. Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*. Association for Computing Machinery (ACM), 2013. `doi:10.1145/2500365.2500592`.
6. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! *Lecture Notes in Computer Science*, pages 147–162, 2013. `doi:10.1007/978-3-319-03545-1_10`.
7. Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, pages 689–700. Association for Computing Machinery (ACM), 2015. `doi:10.1145/2676726.2677006`.
8. Edsger W. Dijkstra. A constructive approach to the problem of program correctness. Circulated privately, August 1967. URL: `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF`.
9. Sebastian Egner, Jeremy Johnson, David Padua, Jianxin Xiong, and Markus Püschel. Automatic derivation and implementation of signal processing algorithms. *SIGSAM Bull.*, 35(2):1–19, June 2001. `doi:10.1145/511988.511990`.
10. Georges Gonthier. Formal proof – the four-color theorem. *Not. ACM*, 55(11):1382–1393, 2008.
11. Jason Gross. An extensible framework for synthesizing efficient, verified parsers. Master's thesis, Massachusetts Institute of Technology, September 2015. URL: `https://people.csail.mit.edu/jgross/personal-website/papers/2015-jgross-thesis.pdf`, `doi:1721.1/101581`.
12. Michael Hammer. The design of usable programming languages. In *Proc. ACM*, ACM'75, pages 225–229, New York, NY, USA, 1975. ACM. `doi:10.1145/800181.810327`.
13. Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proc. PLDI*, PLDI'11, pages 38–49, New York, NY, USA, 2011. ACM. `doi:10.1145/1993498.1993504`.
14. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. ESOP*, volume 213, pages 187–196. Springer Berlin Heidelberg, 1986. `doi:10.1007/3-540-16442-1_14`.
15. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. `doi:10.1007/BF00289507`.
16. E. Horowitz, A. Kemper, and B. Narasimhan. A survey of application generators. *IEEE Softw.*, 2(1):40–54, January 1985. `doi:10.1109/MS.1985.230048`.
17. Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic program-

ming algorithms using solver-aided transformations. In *Proc. OOPSLA*. Association for Computing Machinery (ACM), 2016. doi:10.1145/2983990.2983993.

18 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proc. OOPSLA*, OOPSLA'10, pages 444–463, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869497.

19 Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proc. OOPSLA*, pages 407–426, 2013. doi:10.1145/2509136.2509555.

20 Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974. doi:10.1145/356635.356640.

21 Peter Lammich. Refinement to Imperative/HOL. In *Proc. ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer International Publishing, 2015. doi:10.1007/978-3-319-22102-1_17.

22 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In Lennart Beringer and Amy Felty, editors, *Proc. ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32347-8_12.

23 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proc. DSL*, pages 109–122, 1999. doi:10.1145/331960.331977.

24 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, pages 348–370, 2010. doi:10.1007/978-3-642-17511-4_20.

25 K. Rustan M. Leino and Aleksandar Milicevic. Program extrapolation with Jennisys. In *Proc. OOPSLA*, pages 411–430, 2012. doi:10.1145/2384616.2384646.

26 Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. Association for Computing Machinery (ACM), 2006. doi:10.1145/1111037.1111042.

27 Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proc. VHLL*, pages 50–59, New York, NY, USA, 1974. ACM. doi:10.1145/800233.807045.

28 Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proc. PLDI*, pages 355–368, 2016. doi:10.1145/2908080.2908122.

29 H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, September 1983. doi:10.1145/356914.356917.

30 Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Formal derivation of concurrent garbage collectors. In *Proc. MPC*, pages 353–376, 2010. doi:10.1007/978-3-642-13321-3_20.

31 Clément Pit-Claudel. Compilation using correct-by-construction program synthesis. Master's thesis, Massachusetts Institute of Technology, August 2016. URL: http://pit-claudel.fr/clement/MSc/, doi:1721.1/107293.

32 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proc. POPL*, POPL'13, pages 497–510, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429128.

33 Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proc. POPL*. Association for Computing Machinery (ACM), 1979. doi:10.1145/567752.567771.

34 Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Softw.*, 16(9):1024–1043, September 1990. doi:10.1109/32.58788.

35 Douglas R. Smith and Stephen J. Westfold. Synthesis of propositional satisfiability solvers. Manuscript, 2008.

36 Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proc. PLDI*, PLDI'05, pages 281–294, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065045.

37 Specware. URL: http://www.kestrel.edu/home/prototypes/specware.html.

38 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. PLDI*, PLDI'11, pages 132–141, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993514.

39 Tijs van der Storm, William R. Cook, and Alex Loh. Object grammars: Compositional and bidirectional mapping between text and graphs. In *Proc. SLE*, pages 4–23, 2012. doi:10.1007/978-3-642-36089-3_2.

40 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. doi:10.1145/352029.352035.