Towards Context-Aware Data Refinement

Paul Krogmeier
Purdue University
pkrogmei@purdue.edu

Steven Kidd
Purdue University
kidd9@purdue.edu

Benjamin Delaware
Purdue University
bendy@purdue.edu

Abstract

Fiat is a deductive synthesis framework for deriving correct-by-construction implementations of abstract data types in Coq. The framework uses the representation independence provided by data abstraction to ensure that a derived implementation meets the specification for any possible client. The restriction that an implementation works for *every* client removes potential optimizations that would be correct for a *particular* client, however. The proposed talk discuss our ongoing work on formalizing a relaxation of data refinement in order to enable synthesis of implementations that are tailored to a particular client, while preserving the same representation independence guarantees programmers are used to.

1 Introduction

The ability for users to define their own data abstractions is ubiquitous in modern programming languages. From abstract data types (ADTs) in Clu [11], to classes in Java [5], to typeclasses in Haskell [14], programmers are accustomed to having some mechanism for protecting their code from the details of a particular implementation of an abstract interface. A key property of all these is that the host language enforces the abstraction, providing a contract of representation independence [12] shielding a client from the decisions made by the implementor of a module. This contract also enables implementors to safely apply any optimizations that rely on representation invariants. Fiat [3], a Coq library for deriving correct-by-construction implementations of ADTs, exploits this latter property to automatically derive implementations from specifications using high-level algorithmic and data structure optimizations while ensuring that they are opaque to *any* client.

To concretize matters, consider the simple functional implementation of a list of bytes in Fiat shown in Figure 1. The methods of the ADT allow clients to: create empty ByteStrings, or build them from lists of bytes; add bytes to, and remove them from, the front of a ByteString; and concatenate two ByteStrings together. The definition uses an algebraic datatype for lists to represent the string of bytes. While it leads to a concise specification of the ADT's functionality, this choice is much too inefficient for high-performance applications: the developers of WARP, an http server written in Haskell which makes heavy uses of bytestrings, note that the list structure is "too slow" [1] for their requirements. They instead use an optimized ByteString implementation from the Haskell standard library, which stores data in manually-allocated memory buffers. Using Fiat, we were able to derive an efficient implementation of bytestring with performance comparable to the standard library implementation from a specification similar to that in Figure 1 [15].

Note that the requirements of the data abstraction contract can be *too strong*, however: forcing an implementation to work for *any* client can disallow optimizations that may be sound for a *particular* client, resulting in less performant code. The authors of WARP found the interface of the ByteString library too restrictive in many

```
Definition ByteString := Def ADT {
  RepType := list Word,
  Constructor Empty : rep := [],
  Constructor Pack (xs : list Word) : rep := xs,
  Method Unpack (this : rep) : rep × list Word := (this, this),
  Method Cons (this : rep) (w : Word) : rep := cons w this,
  Method Uncons (this : rep) : rep × option Word :=
    match this with
     | nil ⇒ ([], None)
     | cons x xs ⇒ (xs, Some x)
    end,
  Method Append (this : rep) (r2 : rep) : rep := this ++ r2 }.
```

Figure 1. Specification of ByteString library.

cases. To skirt this problem, WARP directly manipulates the underlying memory buffers, breaking the data abstraction boundary. This removes the protections afforded by the ByteString interface, requiring the developers, not the language, to ensure that invariants on the data structure are never violated. More concretely, Figure 2 shows the Haskell implementations of two functions that store an ascii representation of a number in a bytestring. The first function uses the Cons method from the ByteString interface, while the second is from the WARP implementation and operates directly on the internal ByteString representation. Cons creates a copy of the tail of the bytestring at each invocation, which is unnecessary in this case. The WARP implementation avoids this step by allocating a memory buffer and setting bytes directly using poke, at the cost of potentially overflowing the buffer if the length was calculated incorrectly. Benchmarking shows that the second implementation uses roughly half as much memory as the proper client.

The proposed talk will discuss ongoing work on relaxing the standard notion of data independence with respect to *any* client to one with respect to a *specific* client, in order to synthesize ADT implementations in Fiat that are observationally equivalent from the perspective of that client. The talk will focus on our in-progress development of a core calculus for deductive synthesis of ADT implementations, including the corresponding Coq formalization, a discussion of our notion of context-aware ADT refinement, and our preliminary experiments using context-aware data refinement in Fiat.

```
packIntNaive :: Integral a ⇒ a → ByteString
packIntNaive 0 = Empty
packIntNaive n = Cons (fromIntegral (48 + (mod n 10))) (packIntNaive (div n 10))

packIntWarp :: Integral a ⇒ a → ByteString
packIntWarp 0 = "0"
packIntWarp n = unsafeCreate len go0
  where
    n' = fromInt n + 1 :: Double
    len = ceiling $ logBase 10 n'
    go0 p = go n $ p plusPtr (len − 1)
    go :: Int a ⇒ a → Ptr Word8 → IO ()
    go i p = do
        let (d,r) = i divMod 10
        poke p (48 + fromIntegral r)
        when (d /= 0) $ go d (p plusPtr (−1))
```

Figure 2. Idealized and WARP ByteString clients.

CoqPL'18, , Los Angeles, CA, USA
2018.

2 Formalizing Context-Aware Data Refinement

This section provides more detail on our formalization of a core calculus for Fiat[1]. Figure 3 presents the syntax and selected typing and reduction rules for our calculus. The calculus is a variant of PCF extended with an arbitrary set of algebraic data types T, abstract data types (ADTs), and, most importantly, a nondeterministic choice operator, $\{x : \tau \mid P\,(x, e_1, ..., e_n)\}$. This operator evaluates to any value that satisfies the predicate $P\,(x, e_1, ..., e_n)$. Intuitively, a choice expression precisely spells out *what* is to be computed, but its operational semantics, given in ChoiceR, do not specify *how* to compute it. Programs in this calculus consist of an initial sequence of ADT definitions followed by a client program that calls the operations of those ADTs. The semantics of these operations is defined with respect to a distinguished representation type, **rep**.

We have proven progress and preservation for the calculus in Figure 3. The former requires a slight adjustment to the standard statement, as terms with choice operators may be well typed but unable to reduce. This can happen when the predicate used in a choice expression is uninhabited. To capture this possibility, we update the definition of progress to use a hasChoice predicate over terms:

Theorem 2.1 (Progress). A program that is well typed under an empty environment is either a value, takes a step, or contains a choice operator.

$$\vdash p : \tau \to \text{value } p \;\vee\; \exists p'.p \longrightarrow p' \;\vee\; \text{hasChoice } p$$

We say that a program p_2 *refines* another program p_1 when the possible evaluations of the former are a subset of the latter:

$$p_1 \supseteq p_2 \triangleq \forall v.\; p_2 \longrightarrow v \;\;\to\;\; p_1 \longrightarrow v$$

Definition 2.2 (ADT Refinement). An ADT a_n refines an ADT a_o if every operation of a_n produces a subset of the concrete values produced by that operation in a_o and guarantees similarity of their respective representation types under an abstraction relation $\approx$, when applied to related arguments:

$$\begin{aligned} a_o \supseteq a_n \;\;\equiv\;\; & \exists \approx : \mathbf{rep}_o \to \mathbf{rep}_n \to \text{Prop}.\\ & \forall\, \text{op}\; \overline{r_o}\; \overline{r_n}\; \overline{x}\; r'_n\; v.\; \overline{r_o \approx r_n} \;\to\; a_n.\text{op}(\overline{r_n}, \overline{x}) \longrightarrow (r'_n, v)\\ & \to \exists r'_o.\; a_o.\text{op}(\overline{r_o}, \overline{x}) \longrightarrow (r'_o, v) \;\wedge\; r'_o \approx r'_n \end{aligned}$$

Definition 2.3 (Soundness of Data Refinement). We say that substituting an ADT I_i in a well-formed client program with a refined implementation I'_i is *sound* when it produces a refined program:

$$\begin{array}{lcl} \textbf{let } X_1 := I_1 \textbf{ in } ... & & \textbf{let } X_1 := I_1 \textbf{ in } ...\\ \textbf{let } X_i := I_i \textbf{ in } ... & \supseteq & \textbf{let } X_i := I_i\,' \textbf{ in } ...\\ \textbf{let } X_n := I_n \textbf{ in } e & & \textbf{let } X_n := I_n \textbf{ in } e \end{array}$$

Proposition 2.4 (Representation Independence). ADT refinement guarantees soundness of substitution for *any* client program:

$$\forall I_o I_n . I_o \supseteq I_n \to p \supseteq p[I_o \mapsto I_n]$$

We are currently developing a type system for our refinement calculus which gathers information about the usage of ADT operations via a set of constraints, Ψ. The typing judgement has the form $\Delta;\; \Gamma \vdash_X e \;:\; \tau \mid \Psi$. We plan to use these constraints in an extended definition of ADT refinement that accounts for the usage of an ADT's operations, in order to show a refinement is sound with respect to a particular client.

[1] Our in-progress Coq formalization of this calculus is available at: https://github.com/paulkrog/formalized-fiat.

$$\tau ::= \tau \to \tau \mid X \mid T\;(*\text{ Algebraic Data Types }*) \mid \exists X.\tau$$

$$\begin{aligned} e ::= \;& x \mid C\,(e_1, ..., e_n) \mid e_1 e_2 \mid \textbf{fix } f\,(x : \tau_1) : \tau_2 := e\\ & \mid \textbf{match } e \textbf{ with } \mid C_1(x_1, ..., x_n) \mapsto e_1 \mid ... \mid C_n(x_1, ..., x_n) \mapsto e_n \textbf{end}\\ & \mid \{\, x : \tau \mid P\,(x, e_1, ..., e_n)\,\}\;(*\text{ Choice Operator }*) \end{aligned}$$

$$\begin{aligned} I ::= \;& \text{ADT}\,\{\, \textbf{rep} := \tau_r;\\ & \quad \text{op}_1 := \lambda\,(r_1 \ldots r_n : \textbf{rep})\,(x_1 : \tau_1) \ldots (x_n : \tau_n) : \textbf{rep} \times \tau := e_1;\\ & \quad \ldots\\ & \quad \text{op}_n := \lambda\,(r_1 \ldots r_n : \textbf{rep})\,(x_1 : \tau_1) \ldots (x_n : \tau_n) : \textbf{rep} \times \tau := e_n\\ & \}\text{ as }\{\, \exists X.\, (\overline{X} \to \overline{\tau} \to (X \times \tau)) \times \ldots \times (\overline{X} \to \overline{\tau} \to (X \times \tau))\} \end{aligned}$$

$$p ::= \textbf{let } \{X, x\} := I \textbf{ in } p \mid e$$

$$\frac{}{\Gamma \vdash \textbf{match } C_i(v_1, ..., v_n) \textbf{with } \mid C_1(x_1, ..., x_n) \mapsto e_1 \mid ... \mid C_n(x_1, ..., x_n) \mapsto e_n \textbf{end} \longrightarrow e_i[\overline{x \mapsto v}]} \;(\text{MatchR})$$

$$\frac{\Gamma \vdash P\,(v, v_1, ..., v_n)}{\Gamma \vdash \{x : \tau \mid P\,(x, v_1, ..., v_n)\} \longrightarrow v} \;(\text{ChoiceR})$$

$$\frac{}{\textbf{let } \{X, x\} := I \textbf{ in } p \longrightarrow p[x \mapsto e][X \mapsto \textbf{rep}_I]} \;(\text{ProgLetR})$$

$$\frac{\Delta;\; \Gamma \vdash e : T \qquad C_i : \overline{\tau_i} \to T \qquad \overline{\Delta;\; \Gamma, [\overline{x \mapsto \tau_i}] \vdash e_i : \tau}}{\Delta;\; \Gamma \vdash \textbf{match } e \textbf{ with } \mid C_1(x_1, ..., x_n) \mapsto e_1 \mid ... \mid C_n(x_1, ..., x_n) \mapsto e_n \textbf{end} : \tau} \;(\text{MatchT})$$

$$\frac{\Delta;\; \Gamma \vdash P : \tau \to \tau_1 \to ... \tau_n \to \text{Prop} \qquad \overline{\Delta;\; \Gamma \vdash e_i : \tau_i}}{\Delta;\; \Gamma \vdash \{x : \tau \mid P\,(x, e_1, ..., e_n)\} : \tau} \;(\text{ChoiceT})$$

$$\frac{\Delta;\; \Gamma \vdash I : \exists X.\tau \qquad \Delta, X;\; \Gamma, x : \tau \vdash p : \tau_2}{\Delta;\; \Gamma \vdash \textbf{let } \{X, x\} := I \textbf{ in } p : \tau_2[X \mapsto \textbf{rep}_I]} \;(\text{PLetT})$$

Figure 3. Syntax and select reduction and typing rules for core Fiat calculus.

Proposition 2.5 (Soundness of context-aware ADT refinement). If a program e is well-formed subject to some set of contraints Ψ on ADT I_o, $\Delta;\; \Gamma \vdash_{I_o} e \;:\; \tau \mid \Psi$, and we are able to construct a refined ADT I'_o, subject to those constraints, $\Psi \vdash I_o \supseteq I'_o$, then it is sound to substitute I'_o for I_o in e.

Given a proof of the above proposition, a Fiat derivation of an ADT implementation can take advantage of the information of how ADT operations are called in order to justify more nuanced optimization strategies.

3 Related Work

The concept of deriving implementations that are correct by construction via stepwise refinement has been around since at least the late sixties [4, 16]. Hoare [6] first proposed specifying and verifying algorithms at a high level using abstract data representations which could be transported to more efficient implementations via abstraction functions. Data refinement [13] frameworks exist for both Coq [2] and Isabelle [8–10]. Both frameworks transport proofs about abstract, proof-oriented data representations to more efficient implementations via *unconditional* refinement of data types.

References

[1] Tavish Armstrong. 2013. *The Performance of Open Source Applications*. Lulu.com.

[2] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free! In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Lecture Notes in Computer Science, Vol. 8307. Springer International Publishing, 147–162. https://doi.org/10.1007/978-3-319-03545-1_10

[3] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proc. POPL*.

[4] Edsger W. Dijkstra. 1967. A constructive approach to the problem of program correctness. (Aug. 1967). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF Circulated privately.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2005. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

[6] J. He, C.A.R. Hoare, and J.W. Sanders. 1986. Data refinement refined. In *ESOP 86*, Bernard Robinet and Reinhard Wilhelm (Eds.). Lecture Notes in Computer Science, Vol. 213. Springer Berlin Heidelberg, 187–196.

[7] Iron Lambda [n. d.]. ([n. d.]). http://iron.ouroborus.net/.

[8] Peter Lammich. 2013. Automatic Data Refinement. In *Interactive Theorem Proving*. Springer Berlin Heidelberg.

[9] Peter Lammich. 2015. Refinement to Imperative/HOL. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Lecture Notes in Computer Science, Vol. 9236. Springer International Publishing, 253–269. https://doi.org/10.1007/978-3-319-22102-1_17

[10] Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to HopcroftâĂŹs Algorithm. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Lecture Notes in Computer Science, Vol. 7406. Springer Berlin Heidelberg, 166–182.

[11] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/800233.807045

[12] John C. Mitchell. 1986. Representation Independence and Data Abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*. ACM, New York, NY, USA, 263–276. https://doi.org/10.1145/512644.512669

[13] Carroll Morgan. 1993. The refinement calculus. In *Program Design Calculi*. Springer, 3–52.

[14] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[15] John Wiegley and Benjamin Delaware. 2017. Using Coq to Write Fast and Correct Haskell. In *Proceedings of the 2017 ACM SIGPLAN Symposium on Haskell (Haskell '17)*. ACM, New York, NY, USA.

[16] Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4 (1971), 221–227.